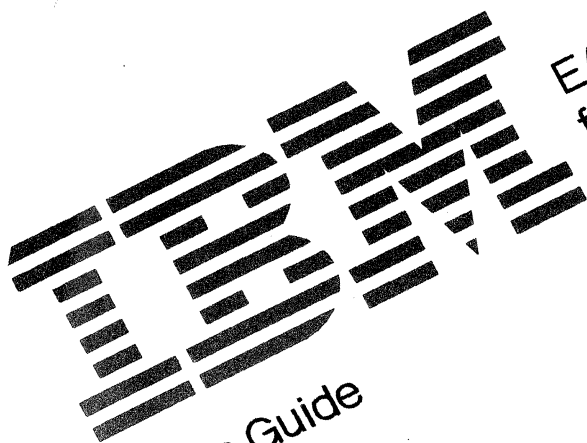




Developer's Guide
Volume 1

EASEL Version 1.1
for OS/2
Extended Edition

IBM PROGRAM LICENSE AGREEMENT

BEFORE OPENING THIS PACKAGE, YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS. OPENING THIS PACKAGE INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THEM YOU SHOULD PROMPTLY RETURN THE PACKAGE UNOPENED AND YOUR MONEY WILL BE REFUNDED.

This is a license agreement and not an agreement for sale. IBM owns, or has licensed from the owner, copyrights in the Program. You obtain no rights other than the license granted you by this Agreement. Title to the enclosed copy of the Program, and any copy made from it, is retained by IBM. IBM licenses your use of the Program in the United States and Puerto Rico. You assume all responsibility for the selection of the Program to achieve your intended results and for the installation of, use of, and results obtained from, the Program.

The Section in the enclosed documentation entitled "License Information" contains additional information concerning the Program and any related Program Services.

LICENSE

You may:

1. use the Program on only one machine at any one time, unless permission to use it on more than one machine at any one time is granted in the License Information (Authorized Use);
2. make a copy of the Program for backup or modification purposes only in support of your Authorized Use. However, Programs marked "Copy Protected" limit copying;
3. modify the Program and/or merge it into another program only in support of your Authorized Use; and
4. transfer possession of copies of the Program to another party by transferring this copy of the IBM Program License Agreement, the License Information, and all other documentation along with at least one complete, unaltered copy of the Program. You must, at the same

time, either transfer to such other party or destroy all your other copies of the Program, including modified copies or portions of the Program merged into other programs. Such transfer of possession terminates your license from IBM. Such other party shall be licensed, under the terms of this Agreement, upon acceptance of this Agreement by its initial use of the Program.

You shall reproduce and include the copyright notice(s) on all such copies of the Program, in whole or in part.

You shall not:

1. use, copy, modify, merge, or transfer copies of the Program except as provided in this Agreement;
2. reverse assemble or reverse compile the Program; and/or
3. sublicense, rent, lease, or assign the Program or any copy thereof.

LIMITED WARRANTY

Warranty details and limitations are described in the Statement of Limited Warranty which is available upon request from IBM, its Authorized Dealer or its approved supplier and is also contained in the License Information. IBM provides a three-month limited warranty on the media for all Programs. For selected Programs, as indicated on the outside of the package, a limited warranty on the Program is available. The applicable Warranty Period is measured from the date of delivery to the original user as evidenced by a receipt.

Certain Programs, as indicated on the outside of the package, are not warranted and are provided "AS IS."

Continued on inside back cover.



Developer's Guide
Volume 1

EASEL Version 1.1
for OS/2
Extended Edition

Second Edition (May 1990)

This publication could include technical inaccuracies or typographical errors. Changes are made periodically to the information herein; these changes will be incorporated in new editions of this publication.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

Products and publications are not stocked at the address given below. Requests for copies of this product and for technical information about the system should be made to your local IBM representative.

A form for reader's comments is provided at the back of this publication. If the form has been removed, address your comments to IBM Corporation, Software Development, Dept. 877, 10401 Fernwood Road, Bethesda, Maryland 20817. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1990. All rights reserved.

© Copyright Interactive Images, Inc. 1989, 1990.

Note to US Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

IBM and OS/2 are registered trademarks of International Business Machines Corporation.

EASEL is a registered trademark of Interactive Images, Inc.

Preface

IBM EASEL[®] for OS/2[®] Extended Edition, which is based on IBM Operating System/2[™] (OS/2) Presentation Manager[™], is a graphical user interface development facility. It is comprised of two licensed programs for developing and executing graphical user interface applications on IBM OS/2 Extended Edition programmable workstations. IBM EASEL for OS/2 Extended Edition (referred to as EASEL OS/2 EE) can support stand-alone applications, the workstation portion of cooperative processing applications, and applications that are front-ends to existing host terminal-based applications.

This publication is meant to help graphical user interface application developers, whose goal is to overcome end-user training and retention time, learn to write EASEL OS/2 EE graphical user interface applications. Although the publication does not assume previous computer graphics experience, you should be familiar with basic programming concepts (such as variables and loops) and IBM OS/2 Extended Edition. Before proceeding, you should be comfortable using simple OS/2 commands (such as TYPE and CD) and an OS/2 text editor.

EASEL OS/2 EE is developed by Interactive Images, Inc., Woburn, MA.

IBM and OS/2 are registered trademarks of International Business Machines Corporation.

EASEL is a registered trademark of Interactive Images, Inc.

Operating System/2 and Presentation Manager are trademarks of International Business Machines Corporation.

This publication is organized as follows:

Chapter	1	Installation
	2	Welcome to EASEL OS/2 EE
	3	Programming with EASEL OS/2 EE
	4	Language Elements
	5	Objects, Coordinate Systems, and Fonts
	6	Responses, Blocks, Action Statements, and Classes
	7	Graphical Objects, Image Regions, and Sense Regions
	8	Textual Regions
	9	Dialog Boxes, Dialog Regions, and Dialog Controls
	10	Region Attributes and Action Bars
	11	Program Flow and Performance
	12	Layout/CUA™
	13	Library Subroutines and Functions
	14	Application Communications, PM Clipboard, and Stimulus
	15	Controlling the Keyboard and Pointing Devices
	16	Business Graphics
	17	SQL Support
	18	Audio Support
	19	APPC Support
	20	3270 Support
	21	5250 Support
	22	Printing and Plotting
	23	Environment Files
	24	Debugging
	25	Writing Local Applications and Libraries
Appendix	A	Include Files
	B	Specified Operating Environment
	C	Fatal Runtime Errors

The accompanying *IBM EASEL Version 1.1 for OS/2 Extended Edition Language Reference Manual* provides a summary of the EASEL OS/2 EE language statements and functions, in alphabetical order. The *IBM EASEL Version 1.1 for OS/2 Extended Edition Quick Reference Card* provides a quick reference to the EASEL OS/2 EE syntax of commonly used language statements and functions.

Layout/CUA is a trademark of Interactive Images, Inc.

Notation Conventions

The syntax models of EASEL OS/2 EE language statements and functions presented in this publication use the conventions shown below.

Convention	Meaning	Examples
<code>^</code>	Indicates a control character.	<code>^D</code>
bold lowercase words	EASEL OS/2 EE statement name or keyword. You must specify exactly as shown.	region selected line is checked
bold symbols	You must specify exactly as shown.	() [] : :
UPPERCASE WORDS	Value or identifier you specify. You must substitute your own identifiers or values.	OBJECT_NAME LOOP_NAME LINE_NO
Square brackets []	Optional entry. You can optionally specify the entry in brackets.	[primary] [column]
	Multiple choice optional entry. You can select one or none of the entries in the brackets. Choices are separated by a vertical bar ().	[guarded resumable]
	When used to specify elements and dimensions in array definitions and references, you must specify exactly as shown.	string ARRAY[N]
Underlined keywords	The default. Does not need to be specified.	[<u>enabled</u> disabled]

Convention	Meaning	Examples
Braces { }	Required entry. You must select one of the entries in the braces. Choices are separated by a vertical bar ().	{ keyboard APPLICATION }
Ellipses ... (horizontal)	Optional repetition. You can repeat the preceding bracketed entry any number of times.	[ACTION_STATEMENT]... [in PARENT_NAME2]...
	When used without a preceding bracketed entry, indicates an omitted portion of syntax.	response to border
: (vertical)	Omitted code. Portions of the EASEL OS/2 EE code have been omitted.	begin Block_Name : end Block_Name

Identifiers and Color Values

In all syntax models, except compile-time definitions, an *identifier* can be replaced with a string value representing the identifier. Compile-time definitions are those definitions not created during runtime with the **add** action statement.

In all syntax models, a compile-time definition *identifier* can be replaced with a compile-time string value representing the identifier.

In all syntax models, COLOR is a string value representing a color keyword, an integer value representing a color number, or a color keyword.

In all syntax models, the keyword **color** preceding COLOR is only optional if COLOR is specified as a color keyword, without quotation marks.

Contents

Chapter 1. Installation	1-1
1.1 Operating Environment	1-2
1.1.1 Machine Requirements	1-2
1.1.2 Program Requirements	1-3
1.1.3 Serial Number	1-3
1.2 Installing EASEL OS/2 EE	1-4
1.2.1 Installing the EASEL OS/2 EE Development System	1-4
1.2.2 Installing Only the EASEL OS/2 EE Runtime Facility	1-5
1.2.3 Diskettes Required	1-6
1.3 Directories Created and Files Added	1-7
1.3.1 Directories Created During Installation	1-7
1.3.2 Files Added During Installation	1-7
1.4 Testing the System	1-12
1.4.1 Testing the EASEL OS/2 EE Runtime	1-12
1.4.2 Testing the EASEL OS/2 EE Compiler	1-12
1.5 Installation Error Messages	1-13
 Chapter 2. Welcome to EASEL OS/2 EE	 2-1
2.1 Why EASEL OS/2 EE?	2-2
2.1.1 Benefits of the EASEL OS/2 EE Language	2-3
2.2 Writing an Event-Driven Application	2-5
2.2.1 EASEL OS/2 EE Front-End Application	2-5
2.3 The Development Process	2-7
2.4 Distributing the Application	2-8
 Chapter 3. Programming with EASEL OS/2 EE	 3-1
3.1 Summary of Changes	3-2
3.1.1 New Features	3-2
3.1.2 Enhancements	3-3
3.2 How an Application is Created	3-5
3.3 Program Construction	3-6
3.3.1 Recommended Order of Program Components	3-6
3.3.2 Using Action Statements	3-7
3.4 Constructing a Program	3-8
3.4.1 An Introduction to Objects and Responses	3-8
3.4.2 Sample Program	3-9
3.4.3 Trying Out EASEL OS/2 EE	3-11
3.5 Creating, Compiling, and Executing an EASEL OS/2 EE Application	3-12
3.5.1 Creating an EASEL OS/2 EE Application	3-12
3.5.2 Compiling an EASEL OS/2 EE OS/2 EE Application	3-12
3.5.3 Executing an EASEL OS/2 EE Application	3-13
3.5.4 Editing the CONFIG.ESL File	3-13
3.6 How EASEL OS/2 EE Compiles and Executes the Program	3-15
3.7 Frame Oriented Syntax	3-16
3.7.1 Screen Size	3-16

3.7.2	Defining Frame Related Attributes	3-17
3.7.3	Changing Title Bar Text	3-18
3.8	Windowed versus Fullscreen Applications	3-19
3.8.1	Overview	3-19
3.8.2	Color Control	3-20
3.8.3	Pointing Devices	3-20
3.8.4	Scaling in a Windowed Application	3-20
3.8.5	Architectural Considerations in a Windowed Application	3-21
3.8.6	Sizing Considerations	3-21
3.8.7	System Menu Accelerators	3-21
3.8.8	Examples	3-21
Chapter 4.	Language Elements	4-1
4.1	The EASEL OS/2 EE Character Set	4-2
4.1.1	Blanks	4-2
4.1.2	Comments	4-3
4.2	National Language Support	4-4
4.2.1	Code Page Support	4-4
4.2.2	Keyboard Support	4-5
4.2.3	Date and Time Support	4-7
4.3	Keywords and Identifiers	4-8
4.4	Values	4-10
4.4.1	Categories and Types of Values	4-10
4.4.2	Compile-time Values	4-11
4.4.3	Literals	4-12
4.4.4	Constants	4-16
4.4.5	Variables	4-17
4.5	Assigning Values to Variables During Runtime	4-20
4.5.1	The copy Action Statement	4-20
4.5.2	The append Action Statement	4-21
4.5.3	The extract from Action Statement	4-22
4.6	Arrays: Defining and Referencing Groups of Constants or Variables	4-37
4.6.1	Specifying Dimensions in an Array Definition	4-37
4.6.2	Specifying Elements in an Array Definition	4-38
4.6.3	Global Variable Arrays	4-39
4.6.4	Referencing Elements in Arrays	4-39
4.6.5	Resizing an Array	4-40
4.7	Built-in Functions	4-42
4.7.1	Response Inquiry Functions	4-42
4.7.2	Object Inquiry Functions	4-43
4.7.3	Item Inquiry Functions	4-44
4.7.4	Action Inquiry Functions	4-44
4.7.5	Special Inquiry Functions	4-45
4.8	Expressions, Operators, and Operands	4-47
4.8.1	Arithmetic Operators	4-48
4.8.2	Relational Operators	4-50
4.8.3	Boolean Operators	4-51
4.9	Type Conversions	4-53
4.9.1	String/Integer and String/Floating Point Conversions	4-53

4.9.2 Boolean/String Conversions	4-54
4.9.3 Integer/Floating Point Conversions	4-54
4.9.4 String/Name Conversions	4-55
4.9.5 String/Name with Ancestry Conversions	4-55
4.10 Modules	4-57
Chapter 5. Objects, Coordinate Systems, and Fonts	5-1
5.1 Types of Objects and Their Uses	5-2
5.2 The Components of an Object	5-3
5.3 Common Object Attributes	5-5
5.3.1 Selectability	5-5
5.3.2 Visibility	5-8
5.3.3 Color	5-8
5.3.4 Ancestry	5-12
5.3.5 Priority	5-14
5.3.6 Parameter	5-15
5.4 Coordinate Systems and Cursors	5-16
5.4.1 The Screen Coordinate System	5-16
5.4.2 The Object Coordinate System	5-18
5.4.3 The Ancestor Coordinate System	5-20
5.4.4 Windows and Viewports: The Coordinate Systems of Regions	5-22
5.4.5 The Graphics and Text Cursors	5-32
5.5 Mixing Object Types in a Window	5-39
5.6 Fonts	5-42
5.6.1 Specifying a Font	5-42
5.6.2 Character Cells	5-43
5.6.3 Portability Across Monitors	5-44
5.6.4 EASEL OS/2 EE Fonts	5-45
5.6.5 Presentation Manager Fonts	5-47
5.6.6 Defining a Font	5-48
5.6.7 Image Fonts	5-49
5.6.8 Scalable Outline Fonts	5-50
5.6.9 Fonts in Textual Regions	5-50
5.7 Redrawing an Object	5-52
5.7.1 Redrawing a Specified Object	5-52
Chapter 6. Responses, Blocks, Action Statements, and Classes	6-1
6.1 About Responses	6-2
6.1.1 Overview of Responses and Their Uses	6-2
6.1.2 Responding to Program Startup	6-3
6.1.3 Responding to Objects	6-4
6.1.4 Responding to Action Bar Items	6-6
6.1.5 Using Response Inquiry Built-in Functions	6-7
6.2 Working with Blocks	6-10
6.2.1 Overview	6-10
6.2.2 Defining a Block	6-11
6.2.3 Nested Blocks	6-12
6.2.4 Leaving (Exiting) a Block	6-13
6.2.5 Responding to a Block Being Left Implicitly	6-16
6.2.6 Responding to Stimuli and Returning to a Specified Block	6-17

6.2.7	Disabling Responses Outside a Block	6-18
6.2.8	Responding to Specified Stimuli and Returning to a Block	6-19
6.2.9	Using the response to start Response Definition Within a Block	6-20
6.2.10	Using the response to char Response Definition in Blocks	6-21
6.3	About Action Statements	6-23
6.3.1	Overview of Actions and Their Uses	6-23
6.3.2	Specifying Action Statements	6-23
6.3.3	Creating, Changing, Clearing, and Deleting Objects	6-24
6.3.4	Changing Attributes of Objects	6-28
6.3.5	Changing the Size and Position of an Object	6-31
6.4	Using Classes of Objects	6-34
6.4.1	Overview	6-34
6.4.2	Defining a Class of Objects	6-34
6.4.3	Defining Responses for a Class of Objects	6-35
6.4.4	Specifying Actions for a Class of Objects	6-36
6.4.5	Adding an Object to a Class	6-37
6.4.6	Deleting an Object from a Class of Objects	6-38
6.4.7	Changing Class Membership	6-38
6.4.8	Using Built-in Functions with Classes	6-39
Chapter 7.	Graphical Objects, Image Regions, and Sense Regions	7-1
7.1	Graphical Objects	7-2
7.1.1	Defining Keys	7-2
7.1.2	Defining Graphical Regions	7-4
7.2	Drawing Statements for Graphical Objects	7-9
7.2.1	Overview	7-9
7.2.2	Placement of Drawing Statements	7-9
7.2.3	Moving the Cursors	7-10
7.2.4	Drawing a Line	7-11
7.2.5	Drawing a Box	7-13
7.2.6	Drawing a Polygon	7-14
7.2.7	Drawing a Circle	7-15
7.2.8	Drawing an Ellipse	7-16
7.2.9	Drawing a Wedge	7-18
7.2.10	Drawing Closed Figures (Shapes)	7-19
7.2.11	Defining and Referencing Patterns	7-30
7.2.12	Displaying Text in Graphical Objects	7-44
7.3	Action Statements for Changing Windows and Viewports in Graphical Regions	7-47
7.3.1	Introduction	7-47
7.3.2	Changing the Viewport Size	7-48
7.3.3	Changing the Viewport Position	7-50
7.3.4	Changing the Window Size	7-51
7.3.5	Changing the Window Position	7-52
7.4	Using Built-in Functions with Graphical Objects	7-54
7.4.1	Introduction	7-54
7.4.2	Using Functions to Inquire about Size and Position	7-54
7.4.3	Using Built-in Functions to Inquire about an Object's Attributes	7-59

7.5 Image Regions	7-63
7.5.1 Bitmap File Formats Supported	7-63
7.5.2 Operations Supported	7-64
7.5.3 Defining Image Regions	7-64
7.5.4 Foreground and Background Colors	7-65
7.5.5 Cache Specifications	7-65
7.5.6 Scaling	7-67
7.5.7 Specifying File Contents	7-68
7.5.8 Specifying Image Maps in Image Regions	7-69
7.5.9 Read/Write Statements	7-70
7.6 Sense Regions	7-72
7.6.1 Defining Sense Regions	7-72
7.6.2 The xcoord and ycoord Built-in Functions	7-73
7.6.3 PM Frame Attributes	7-73
Chapter 8. Textual Regions	8-1
8.1 Defining and Using Textual Regions	8-2
8.1.1 Overview	8-2
8.1.2 Defining a Textual Region	8-3
8.2 Drawing Statements for Textual Regions	8-7
8.2.1 Overview	8-7
8.2.2 Moving the Text Cursor	8-9
8.2.3 Defining Text Segments and Text Blocks	8-10
8.2.4 Inserting Text	8-14
8.2.5 Overwriting Text	8-20
8.3 Response Definitions for Textual Regions	8-26
8.4 Action Statements for Textual Regions	8-27
8.4.1 Overview	8-27
8.4.2 Adding Drawing Statements to a Textual Region	8-28
8.4.3 Removing Text	8-28
8.4.4 Controlling Text Emphasis	8-29
8.4.5 Changing the Visibility of the Text Cursor	8-30
8.4.6 Writing a Textual Region to a File	8-30
8.4.7 Reading a File into a Textual Region	8-32
8.4.8 Finding a String of Text in a Textual Region	8-34
8.4.9 Changing the Size and Position of a Textual Region's Window and Viewport	8-36
8.5 Using Built-in Functions with Textual Regions	8-41
8.5.1 Overview	8-41
8.5.2 Using the textual Built-in Functions	8-43
8.5.3 Using the column size of Built-in Function	8-46
8.5.4 Using the line size of Built-in Function	8-47
8.5.5 Using the foreground at and background at Built-in Functions	8-47
8.5.6 Using the found Built-in Function	8-48
8.6 Sample Programs Using Textual Regions	8-49
8.6.1 Sample #1	8-49
8.6.2 Sample #2	8-50
8.7 Working with Colored Textual Regions	8-54
8.7.1 Defining a Colored Textual Region	8-54
8.7.2 Coloring Text That Already Exists in the Program	8-55

Chapter 9. Dialog Boxes, Dialog Regions, and Dialog Controls	9-1
9.1 Dialog Boxes	9-2
9.1.1 Defining Dialog Boxes	9-2
9.1.2 Attributes for Dialog Boxes	9-3
9.1.3 Actions for Dialog Boxes	9-3
9.1.4 Keyboard and Mouse Use in Dialog Boxes	9-3
9.1.5 Example of a Dialog Box	9-4
9.1.6 Built-in Functions for Dialog Boxes	9-4
9.2 Dialog Regions	9-5
9.2.1 Actions for Dialog Regions	9-6
9.2.2 Keyboard and Mouse Use in Dialog Regions	9-6
9.2.3 Built-in Functions for Dialog Regions	9-6
9.3 Dialog Control Objects	9-7
9.3.1 Overview	9-7
9.3.2 Groups	9-7
9.3.3 Mnemonics	9-7
9.3.4 Built-in Functions for Dialog Controls	9-8
9.3.5 Single-Line Entry Fields	9-8
9.3.6 Multiple-Line Entry Fields	9-9
9.3.7 List Boxes	9-11
9.3.8 Combination Boxes	9-15
9.3.9 Drop-down Lists	9-16
9.3.10 Push Buttons	9-17
9.3.11 Radio Buttons	9-18
9.3.12 Check Boxes	9-20
9.3.13 Static Text	9-21
9.3.14 Group Boxes	9-23
9.4 Sample Code for Generating a Dialog Box	9-24
Chapter 10. Region Attributes and Action Bars	10-1
10.1 Region Attributes	10-2
10.1.1 Using the Border Attribute	10-4
10.1.2 Using the Title Bar Attribute	10-4
10.1.3 Using the Minimize Attribute	10-4
10.1.4 Using the Maximize Attribute	10-5
10.1.5 Using the System Menu Attribute	10-5
10.1.6 Using the Scroll Attributes	10-6
10.1.7 Using the Action Bar Attribute	10-6
10.1.8 Using the No Scale Attribute	10-7
10.1.9 Using the Pointer Attribute	10-7
10.2 Region Actions	10-8
10.3 Creating Action Bars	10-9
10.3.1 Defining an Action Bar Template	10-10
10.3.2 Defining Pulldowns	10-11
10.3.3 Defining Choices	10-12
10.3.4 Defining Separators	10-13
10.3.5 Defining Buttons	10-13
10.3.6 Defining Global Items	10-14

10.3.7 Accelerator Keys	10-14
10.3.8 Mnemonics	10-15
10.3.9 Item Parameters	10-16
10.3.10 An Example Action Bar Template	10-17
10.4 Manipulating Action Bars and Items	10-19
10.4.1 Manipulating Action Bars	10-19
10.4.2 Manipulating Action Bar Items	10-20
10.5 Responding to Action Bar Items	10-27
10.5.1 Responding to Items in a Region	10-27
10.6 Built-in Functions for Action Bar Items	10-28
10.6.1 Built-in Functions in Inquiries	10-28
10.6.2 Built-in Functions in Responses	10-29
Chapter 11. Program Flow and Performance	11-1
11.1 Controlling Program Flow	11-2
11.2 Functions, Subroutines, and Action Routines	11-4
11.2.1 Declaring External Subroutines and External Functions	11-4
11.2.2 Action Routines and Internal EASEL OS/2 EE Subroutines	11-4
11.3 General Program Control	11-8
11.3.1 Including Another EASEL OS/2 EE File in the Current Program	11-8
11.3.2 Exiting the Program	11-9
11.3.3 Invoke	11-10
11.3.4 Transferring to Another Program	11-11
11.3.5 Saving a Program in Progress	11-12
11.4 Global and Local Entities	11-14
11.4.1 Variables	11-14
11.4.2 Applications	11-14
11.4.3 Devices	11-14
11.4.4 Memory	11-15
11.4.5 Global Entities in Changed Programs	11-15
11.4.6 Global and Local Values in Saved Programs	11-15
11.5 Controlling Flow Within a Program	11-17
11.5.1 Referencing an Action Routine	11-17
11.5.2 Specifying a Conditional Action Statement	11-18
11.5.3 Specifying a Call Action Statement	11-21
11.5.4 Specifying a for or while loop	11-21
11.5.5 for each member loop	11-23
11.5.6 for each selected line loop	11-24
11.5.7 Exiting from a Loop	11-24
11.5.8 Nested Loops	11-25
11.5.9 Blocks and Loops	11-26
11.5.10 Switch Statement	11-26
11.6 Program Timing	11-29
11.6.1 Responding to Elapsed Time	11-29
11.6.2 Responding to Elapsed Time During Which No Stimulus is Received	11-30
11.6.3 Action Statement to Suspend Processing	11-31
11.7 Runtime Memory Handling	11-33
11.7.1 Global Memory	11-33

11.7.2	Changing Memory Allocation in the EASEL.CMD File	11-34
11.7.3	Controlling Memory from within the Program	11-34
Chapter 12.	Layout/CUA	12-1
12.1	Before You Begin	12-2
12.1.1	What You Need to Use Layout/CUA	12-2
12.1.2	What You Need to Know	12-2
12.1.3	About CUA Conformance	12-2
12.1.4	About This Chapter	12-3
12.2	Using Layout/CUA	12-4
12.2.1	The Layout/CUA Primary Window	12-4
12.2.2	Using the Help System	12-5
12.2.3	Creating an Interface Primary Window	12-7
12.2.4	Creating Secondary Windows	12-8
12.2.5	Creating Dialog Boxes	12-8
12.2.6	Selecting a Tool from the Palette	12-9
12.2.7	Adding Controls to a Dialog Box or Dialog Region	12-11
12.2.8	Selecting Controls Inside a Dialog Box or Dialog Region	12-11
12.2.9	Deleting Objects	12-12
12.2.10	Moving Objects	12-12
12.2.11	Resizing Objects and Groups	12-12
12.2.12	Aligning Objects	12-13
12.2.13	Modifying Object Information	12-13
12.2.14	Editing the Action Bar	12-14
12.2.15	Editing a Pulldown	12-14
12.2.16	Undoing Your Last Action	12-14
12.2.17	Hiding and Showing Windows	12-14
12.2.18	Creating Selection Fields	12-15
12.2.19	Using the Clipboard	12-16
12.2.20	Converting the Interface to EASEL OS/2 EE Code	12-17
12.2.21	Saving the Interface	12-19
12.2.22	Editing an Interface File	12-20
12.3	Code Generation	12-21
12.3.1	Dialog Box Subroutines	12-21
12.3.2	Automatic Dialog-Box Calling	12-23
12.3.3	Pulldown Choices	12-23
12.3.4	Push Buttons	12-23
12.4	Object Information	12-24
12.4.1	Primary Window Info	12-24
12.4.2	Action Bar Choice Info	12-25
12.4.3	Pulldown Choice Info	12-25
12.4.4	Secondary Window Info	12-27
12.4.5	Dialog Box Info	12-28
12.4.6	Push Button Info	12-29
12.4.7	Radio Button Info	12-30
12.4.8	Check Box Info	12-31
12.4.9	List Box Info	12-32
12.4.10	Entry Field Info	12-33
12.4.11	Static Text Info	12-34
12.4.12	Group Box Info	12-34

12.4.13 Multiline Entry Field Info	12-35
12.4.14 Combination Box Info	12-36
Chapter 13. Library Subroutines and Functions	13-1
13.1 Overview	13-2
13.1.1 Include File	13-2
13.1.2 Library Subroutines	13-2
13.1.3 Library Functions	13-3
13.2 The EASEL OS/2 EE Library Functions	13-4
13.2.1 EslQueryEgaColor() Function	13-4
13.2.2 EslSetEgaColor() Function	13-4
13.2.3 EslQueryFocus() Function	13-5
13.3 The Date Library Functions	13-7
13.3.1 DaySpan() Function	13-7
13.3.2 LocalDate() Function	13-8
13.3.3 LocalTime() Function	13-8
13.3.4 Validdate() Function	13-9
13.3.5 WeekDay() Function	13-10
13.4 The Math Library Functions and Subroutines	13-11
13.4.1 The Math Library Functions	13-11
13.4.2 The Math Library Subroutines	13-12
13.4.3 Error Handling	13-13
13.4.4 ABS() Function	13-14
13.4.5 ARCCOS() Function	13-14
13.4.6 ARCSIN() Function	13-15
13.4.7 ARCTAN() Function	13-15
13.4.8 ARCTAN2() Function	13-15
13.4.9 CEIL() Function	13-16
13.4.10 COS() Function	13-16
13.4.11 E() Function	13-17
13.4.12 EXP() Function	13-17
13.4.13 FLOOR() Function	13-17
13.4.14 LN() Function	13-18
13.4.15 LOG() Function	13-18
13.4.16 MOD() Function	13-18
13.4.17 PI() Function	13-19
13.4.18 POWER() Function	13-19
13.4.19 SIN() Function	13-20
13.4.20 SQRT() Function	13-20
13.4.21 TAN() Function	13-20
13.4.22 MAX() Subroutine	13-21
13.4.23 MAXINT() Subroutine	13-21
13.4.24 MEAN() Subroutine	13-21
13.4.25 MEANINT() Subroutine	13-22
13.4.26 MEDIAN() Subroutine	13-22
13.4.27 MEDIANINT() Subroutine	13-22
13.4.28 MIN() Subroutine	13-23
13.4.29 MININT() Subroutine	13-23
13.4.30 STDDEV() Subroutine	13-23
13.4.31 STDDEVINT() Subroutine	13-24

13.4.32 SUM() Subroutine	13-24
13.4.33 SUMINT() Subroutine	13-24
13.4.34 VARIANCE() Subroutine	13-25
13.4.35 VARIANCEINT() Subroutine	13-25
13.5 The File I/O Library	13-26
13.5.1 Overview	13-26
13.5.2 The File I/O Library Subroutines	13-26
13.5.3 The Record Definition	13-27
13.5.4 The Open() and Close() Subroutines	13-31
13.5.5 Input Subroutines	13-35
13.5.6 Output Subroutines	13-41
13.5.7 Parameter Adjustment Subroutines	13-44
13.5.8 Error Handling Subroutine	13-47
13.6 The Message Library Function	13-49

Chapter 14. Application Communications, PM Clipboard, and Stimulus	14-1
14.1 Overview of Applications	14-2
14.1.1 Application Types: Remote and Local	14-2
14.1.2 Defining, Starting, and Stopping Applications	14-3
14.2 Sending and Receiving Data	14-7
14.2.1 Sending Information	14-7
14.2.2 Responding to Data Received from an Application	14-7
14.2.3 How response to line and response to char Are Processed	14-13
14.2.4 Changing the Disposition of Text Input	14-16
14.3 Remote Application Support for EASEL OS/2 EE	14-18
14.3.1 Starting Remote Applications	14-18
14.3.2 Connection Types	14-18
14.3.3 Establishing a Connection	14-20
14.3.4 ACDI Remote Characteristics Keywords	14-21
14.3.5 Failure Codes	14-24
14.3.6 Errorlevel Built-in Function	14-24
14.4 Sample Programs	14-26
14.4.1 Sample Program Using a Local and Remote Application	14-26
14.4.2 Sample Program Using Characters Received from the Keyboard	14-27
14.5 Using the Presentation Manager Clipboard	14-28
14.6 Stimulus Libraries	14-29
14.6.1 Declaring a Stimulus	14-29
14.6.2 Defining a Stimulus Response	14-29
14.6.3 Response Inquiry Functions	14-30

Chapter 15. Controlling the Keyboard and Pointing Devices	15-1
15.1 About Pointing Devices	15-2
15.1.1 Mouse	15-2
15.1.2 Touchscreen	15-2
15.1.3 Mouse Pointer Appearance	15-2
15.1.4 Touchscreen Support	15-3
15.2 Controlling Availability of Pointing Devices	15-6
15.2.1 Using the Default Pointing Device	15-6
15.3 Opening and Closing a Pointing Device	15-7

15.3.1 Controlling Input from a Pointing Device	15-7
15.4 Handling Characters Received from the Keyboard	15-9
15.4.1 Matching Special Characters from the Keyboard	15-9
15.5 Providing Feedback to Selections	15-14
15.5.1 Providing Audio Feedback to Selections	15-14
15.5.2 Providing Video Feedback to Selections	15-14
Chapter 16. Business Graphics	16-1
16.1 Introduction	16-2
16.2 Creating a Graph	16-3
16.2.1 Required Procedures	16-3
16.2.2 Action Routines for Each Graph Type	16-5
16.2.3 Specifying Data Elements for a Graph	16-6
16.2.4 Specifying the Number of Data Sets and Data Elements in a Graph	16-8
16.2.5 Specifying the Decimal Point Character	16-8
16.3 Rules and Variables for Certain Graph Types	16-10
16.3.1 Overlapping Bar Graph	16-10
16.3.2 Three-Dimensional (3-D) Bar Graph	16-10
16.3.3 3-D Outline Bar Graph	16-11
16.3.4 Pie Graph	16-11
16.3.5 Stepped Line Graph	16-13
16.3.6 Stacked Line Graph	16-13
16.4 Customizing a Graph	16-14
16.4.1 Graph Titles	16-15
16.4.2 Axis Titles	16-16
16.4.3 Axis Boundaries	16-19
16.4.4 Tick and Pie Wedge Labels	16-20
16.4.5 Legends	16-24
16.4.6 Footnotes	16-26
16.4.7 Colors	16-27
16.4.8 Coloring Text in a Graph (Reference)	16-29
16.4.9 Grid Lines	16-29
16.4.10 Marker Lines	16-29
16.4.11 Line Widths	16-32
16.4.12 Symbols	16-33
16.4.13 Superimposing Two Graphs	16-34
16.4.14 GMovables: Moving Text in a Graph	16-34
16.5 Sample Program Using the BGRAPH Module	16-37
16.6 Illustrations of Graph Types	16-40
16.7 Summary of BGRAPH Module Variables	16-47
16.8 Public File of the BGRAPH Module	16-50
Chapter 17. SQL Support	17-1
17.1 Introduction	17-2
17.1.1 Requirements	17-2
17.1.2 Access Plans	17-2
17.1.3 Types of Commands	17-2
17.1.4 Input and Output	17-2
17.2 Direct Commands	17-3

17.2.1	startdb Direct Command	17-4
17.2.2	stopdb Direct Command	17-5
17.2.3	binddb Direct Command	17-6
17.3	Configuration Commands	17-7
17.3.1	prompt Configuration Command	17-8
17.3.2	error Configuration Command	17-9
17.3.3	null Configuration Command	17-10
17.3.4	info Configuration Command	17-11
17.3.5	bindfile Configuration Command	17-12
17.3.6	output Configuration Command	17-13
17.3.7	fs Configuration Command	17-14
17.3.8	rs Configuration Command	17-15
17.3.9	set Configuration Command	17-16
17.3.10	maxwidth Configuration Command	17-17
17.3.11	startq Configuration Command	17-18
17.3.12	verbose Configuration Command	17-19
17.3.13	exit Configuration Command	17-20
17.4	Inquiry Commands	17-21
17.4.1	display Inquiry Command	17-22
17.4.2	names Inquiry Command	17-23
17.4.3	types Inquiry Command	17-24
17.4.4	widths Inquiry Command	17-25
17.5	Dynamic SQL Commands	17-26
17.6	Error Handling	17-27
17.6.1	Error Detected by Local Application	17-27
17.6.2	Error Detected by OS/2 Extended Edition Database Manager	17-27
17.7	Command Line Options	17-30
17.8	Notes on Using SQL Support	17-31
17.8.1	Customizing Strings	17-31
17.8.2	Using Output as an Input Command	17-31
17.8.3	LONG VARCHAR Allocation	17-31
17.8.4	Increasing Memory	17-32
17.8.5	Requirement for Dynamic Commands	17-32
17.8.6	Statement Size	17-32
17.8.7	Case and White Space	17-32
17.8.8	Shared Mode	17-32
17.8.9	Isolation Level	17-32
17.8.10	Standard Output and Standard Error	17-32
17.8.11	Determining Number of Records	17-32
17.8.12	Error Message	17-32
17.8.13	Graphic Data Types	17-33
Chapter 18.	Audio Support	18-1
18.1	Overview	18-2
18.1.1	Digitized Audio Files	18-2
18.1.2	Audio Utility	18-3
18.1.3	Include File	18-3
18.1.4	Audio Dynamic Link Library	18-4
18.1.5	Responding to Audio Events	18-4

18.2 EASEL OS/2 EE Audio Support Subroutines	18-6
18.2.1 Setting Argument Values	18-7
18.2.2 Return Codes	18-7
18.2.3 AUDClose()	18-9
18.2.4 AUDErase()	18-10
18.2.5 AUDOpen()	18-11
18.2.6 AUDPause()	18-12
18.2.7 AUDPlay()	18-13
18.2.8 AUDQueryBusy()	18-17
18.2.9 AUDQueryElapsedTime()	18-19
18.2.10 AUDQueryPlayingTime()	18-20
18.2.11 AUDRecord()	18-21
18.2.12 AUDResume()	18-25
18.2.13 AUDSetBalance()	18-26
18.2.14 AUDSetVolume()	18-28
18.2.15 AUDStop()	18-29
18.3 Sample Program Using Audio Support	18-31
18.4 Audio Support Reserved Constants	18-33
Chapter 19. APPC Support	19-1
19.1 Overview	19-2
19.1.1 The Automatic Interface	19-3
19.1.2 The Controlled Interface	19-4
19.1.3 Responding to APPC Events	19-4
19.1.4 APPC Dynamic Link Library	19-5
19.1.5 Include File	19-5
19.1.6 Confirmation Processing	19-5
19.1.7 Forcing a Send	19-6
19.1.8 How EASEL OS/2 EE APPC Support Works	19-6
19.2 EASEL OS/2 EE APPC Events	19-8
19.3 EASEL OS/2 EE APPC Subroutines	19-11
19.3.1 Setting/Querying Profile Information	19-12
19.3.2 The Automatic Interface	19-14
19.3.3 Sending/Receiving Non-string Data	19-15
19.3.4 Confirmation Processing	19-16
19.3.5 The Controlled Interface	19-18
19.3.6 Utility Subroutines	19-22
19.4 Using APPC Subroutines	19-23
19.4.1 Setting Argument Values	19-23
19.4.2 Return Codes	19-24
19.5 Detailed Subroutine Descriptions	19-26
19.5.1 APPCAcceptStart()	19-26
19.5.2 APPCConvertA2E()	19-28
19.5.3 APPCConvertE2A()	19-31
19.5.4 APPCEnd()	19-34
19.5.5 APPCFlush()	19-36
19.5.6 APPCGetAcceptProfile()	19-38
19.5.7 APPCGetAcceptProfileNames()	19-41
19.5.8 APPCGetDLLData()	19-43
19.5.9 APPCGetInfo()	19-46

19.5.10	APPCGetInitProfile()	19-50
19.5.11	APPCGetInitProfileNames()	19-53
19.5.12	APPCGetString()	19-55
19.5.13	APPCGetSystemError()	19-57
19.5.14	APPCReadyToReceive()	19-61
19.5.15	APPCReceiveNotify()	19-63
19.5.16	APPCRequestConfirm()	19-65
19.5.17	APPCRequestToSend()	19-67
19.5.18	APPCSendConfirmed()	19-69
19.5.19	APPCSendDLLData()	19-71
19.5.20	APPCSendError()	19-75
19.5.21	APPCSendString()	19-77
19.5.22	APPCSetAcceptProfile()	19-81
19.5.23	APPCSetInitProfile()	19-84
19.5.24	APPCStart()	19-88
19.5.25	APPCTestReqToSend()	19-91
19.6	Sample Code: Automatic Interface	19-93
19.7	Sample Code: Controlled Interface	19-95
19.8	EASEL OS/2 EE APPC Reserved Constants	19-98
19.9	Communications Manager Considerations	19-100
19.9.1	Configuring CM for an EASEL OS/2 EE Acceptor	19-100
19.9.2	Enabling APPC Tracing	19-101
Chapter 20.	3270 Support	20-1
20.1	Introduction	20-2
20.1.1	EASEL OS/2 EE in the 3270 Environment	20-2
20.1.2	Components of 3270 Support for EASEL OS/2 EE	20-2
20.1.3	Using the M3270 Module: Required Procedures	20-3
20.1.4	Referencing the M3270 Module	20-3
20.1.5	Sample Program	20-4
20.2	3270 Session and Screen Concepts	20-5
20.2.1	Logical Terminal Sessions	20-5
20.2.2	Screen Sizes	20-6
20.2.3	Field-oriented and Line-oriented Screens	20-6
20.3	Retrieving Field Information with FIELDS.EXE	20-8
20.3.1	FIELDS.EXE Sample Output	20-10
20.4	M3270 Module Action Routines	20-13
20.4.1	M3270 Action Routines and EASEL OS/2 EE Built-in Functions	20-13
20.4.2	Summary of M3270 Action Routines	20-14
20.4.3	Starting and Stopping 3270 Support for EASEL OS/2 EE	20-17
20.4.4	Connecting and Disconnecting Sessions	20-19
20.4.5	Examining the Buffer	20-21
20.4.6	Synchronizing EASEL OS/2 EE with the Host	20-33
20.4.7	Communicating with the Host	20-45
20.4.8	Controlling the Emulator Window	20-50
20.4.9	Miscellaneous Action Routines	20-56
20.5	Error Handling	20-64
20.5.1	Error Status Variables	20-64
20.5.2	Error Messages (Cross Reference)	20-66

20.6 Reserved Variables (Cross Reference)	20-71
20.7 Reserved Constants	20-74
20.8 Sample Program Using 3270 Screens	20-75
20.8.1 Sample Program Using EASEL OS/2 EE 3270 Support	20-77
Chapter 21. 5250 Support	21-1
21.1 Introduction	21-2
21.1.1 EASEL OS/2 EE in the 5250 Environment	21-2
21.1.2 Components of 5250 Support for EASEL OS/2 EE	21-2
21.1.3 Using the M5250 Module: Required Procedures	21-3
21.1.4 Referencing the M5250 Module	21-3
21.1.5 Sample Program	21-4
21.2 5250 Session and Screen Concepts	21-5
21.2.1 Session Names	21-5
21.2.2 Session States	21-5
21.2.3 Field-oriented and Line-oriented Screens	21-6
21.3 Retrieving Field Information with FIELDS52.EXE	21-9
21.3.1 Reading the FIELDS52.EXE Report	21-9
21.3.2 Using FIELDS52.EXE	21-9
21.3.3 FIELDS52.EXE Sample Output	21-11
21.4 M5250 Module Action Routines	21-13
21.4.1 M5250 Action Routines and EASEL OS/2 EE Built-in Functions	21-13
21.4.2 Summary of M5250 Action Routines	21-14
21.4.3 Starting and Stopping 5250 Support for EASEL OS/2 EE	21-17
21.4.4 Connecting and Disconnecting Sessions	21-19
21.4.5 Examining the Buffer	21-21
21.4.6 Synchronizing 5250 Support for EASEL OS/2 EE with the Host	21-33
21.4.7 Synchronizing with AS/400 Applications	21-34
21.4.8 Synchronizing with Applications Via 3270 Device Emulation	21-37
21.4.9 Communicating with the Host	21-49
21.4.10 Miscellaneous Action Routines	21-55
21.5 Error Handling	21-60
21.5.1 Error Status Variables	21-60
21.5.2 Error Messages (Cross Reference)	21-62
21.6 Reserved Variables (Cross-Reference)	21-67
21.7 Reserved Constants	21-70
21.8 Sample Program	21-71
21.8.1 Sample Program Using 5250 Support for EASEL OS/2 EE	21-73
Chapter 22. Printing and Plotting	22-1
22.1 Printing and Plotting EASEL OS/2 EE Regions	22-2
22.1.1 About Output Devices	22-2
22.1.2 Overview of Hardcopy Support	22-2
22.1.3 Using the Plot Action Statement	22-3
22.2 Printing ASCII Files Within EASEL OS/2 EE	22-5
22.2.1 Using the Printing Utilities	22-5
22.2.2 Optional Parameters	22-5
22.2.3 Example	22-6

22.2.4 Error Messages	22-7
Chapter 23. Environment Files	23-1
23.1 Overview	23-2
23.2 COMPILE.CMD OS/2 Command File	23-3
23.2.1 Command Line Options	23-3
23.3 CONFIG.ESL File	23-5
23.3.1 Configuration Options	23-5
23.4 EASEL.CMD OS/2 Command File	23-9
23.4.1 Command Line Options	23-9
Chapter 24. Debugging	24-1
24.1 Overview	24-2
24.1.1 Using the Errorlog File	24-2
24.1.2 Reading Errors on Another Terminal	24-2
24.2 Action Statements and Environment Declarations for Debugging	24-3
24.2.1 Tracing Action Statements for Portions of an EASEL OS/2 EE Program	24-3
24.2.2 Sending Values to the Errorlog	24-3
24.3 Command Line Options for Debugging	24-5
24.3.1 Tracing Applications Communications	24-5
24.3.2 Tracing Action Statements for an Entire EASEL OS/2 EE Program	24-5
Chapter 25. Writing Local Applications and Libraries	25-1
25.1 Writing Well-behaved Local Applications	25-2
25.1.1 Standalone Program	25-2
25.1.2 Keyboard and Screen I/O	25-2
25.1.3 Guidelines for Writing C Local Applications for EASEL OS/2 EE	25-2
25.2 Writing Library Functions and Subroutines	25-3
25.2.1 Declaring C Routines in an EASEL OS/2 EE Program	25-3
25.2.2 Function Arguments and Return Values	25-4
25.2.3 Subroutine Arguments and Return Values	25-4
25.2.4 Strings	25-5
25.2.5 C Coding Considerations	25-7
25.2.6 Example Library and Usage	25-8
25.3 Writing a Stimulus Library	25-18
Appendix A. Include Files	A-1
A.1 Overview	A-2
A.2 accel.inc file	A-3
A.3 datelib.inc file	A-4
A.4 egacolor.inc File	A-5
A.5 eslappc.inc file	A-6
A.6 eslaudio.inc file	A-12
A.7 esllib.inc file	A-14
A.8 fileio.inc File	A-15
A.9 mathlib.inc File	A-16
A.10 message.inc File	A-18

A.11 pmptr.inc File	A-20
Appendix B. Specified Operating Environment	B-1
B.1 Machine Requirements	B-2
B.2 Program Requirements	B-4
B.3 Graphics Display Considerations	B-5
B.3.1 Display Resolution	B-5
B.3.2 Portability	B-5
Appendix C. Fatal Runtime Errors	C-1
C.1 Overview	C-2
C.2 Error Messages	C-3
C.2.1 ESL1002	C-3
C.2.2 ESL1003	C-3
C.2.3 ESL1004	C-3
C.2.4 ESL1005	C-3
C.2.5 ESL1006	C-4
C.2.6 ESL1007	C-4
C.2.7 ESL1008	C-4
C.2.8 ESL1009	C-4
C.2.9 ESL1010	C-5
C.2.10 ESL1011	C-5
C.2.11 ESL1012	C-5
C.2.12 ESL1012	C-5
C.2.13 ESL1012	C-5
Index	X-1

Chapter 1. Installation

1.1 Operating Environment

1.1.1 Machine Requirements

IBM EASEL® Version 1.1 for OS/2® Extended Edition (referred to as EASEL OS/2 EE) is designed to execute with the following equipment:

- One of the following system units:
 - IBM Personal Computer XT™ Model 286
 - IBM Personal Computer AT®
 - IBM Personal System/2® Model 30 286, 50, 50Z, 60, 70 386, or 80 386
- 1MB of available memory above OS/2 requirements.
- One high capacity diskette drive.
- Fixed disk drive with 3.5MB of free space.
- Mouse pointing device supported by IBM Operating System/2. (Optional)
- IBM Enhanced Graphics Adapter (EGA), Video Graphics Array (VGA), or 8514/A graphics display adapter and associated IBM display.

Note: IBM Video Graphics Array graphics display adapter is provided as an integrated feature of the IBM Personal System/2 (PS/2®) system unit or by the IBM Personal System/2 Display Adapter.
- Communications adapters supported by IBM OS/2 Extended Edition Communications Manager, which provide LU 6.2, 3270, 5250, or asynchronous communications or remote database services. (Optional)
- IBM Audio Capture and Playback Adapter or IBM Audio Capture and Playback Adapter/A, with audio input and output devices. (Optional)

See B.1, “Machine Requirements” on page B-2 for a detailed description of EASEL OS/2 EE machine requirements.

IBM, OS/2, Personal Computer AT, Personal System/2, and PS/2 are registered trademarks of International Business Machines Corporation.

EASEL is a registered trademark of Interactive Images, Inc.

Personal Computer XT is a trademark of International Business Machines Corporation.

1.1.2 Program Requirements

EASEL OS/2 EE is designed to execute on IBM Operating System/2™ Extended Edition Version 1.2 or later.

See B.2, "Program Requirements" on page B-4 for optional EASEL OS/2 EE program requirements.

1.1.3 Serial Number

The EASEL OS/2 EE Runtime (EASEL.CMD) and Layout/CUA™ (ESLCUA.EXE) display the EASEL OS/2 EE unique serial number in the banner and logo window, respectively. The Runtime banner is written out to wherever the **-e** FILENAME command line option writes to. For example, entering `easel welcome -e banner` at the OS/2 command prompt writes the banner to the file BANNER. Once the application has completed execution, the banner can be viewed by entering type FILENAME at the OS/2 command prompt, where FILENAME is the same name specified with the **-e** command line option.

The Layout/CUA logo window is displayed the first time the user invokes the Layout/CUA tool after installation. After that, it is displayed according to the user's Logo Display setting in the Presentation Manager™ Control Panel Preferences, or when the About... choice in the Help pulldown is selected.

Operating System/2 and Presentation Manager are trademarks of International Business Machines Corporation.

Layout/CUA is a trademark of Interactive Images, Inc.

1.2 Installing EASEL OS/2 EE

1.2.1 Installing the EASEL OS/2 EE Development System

Installation of the EASEL OS/2 EE Development System adds to your fixed disk drive all the applications, utilities, libraries, and files provided with EASEL OS/2 EE for developing EASEL OS/2 EE graphical user interface applications. Installing the EASEL OS/2 EE Development System also includes installation of the EASEL OS/2 EE Runtime Facility.

Before you start the installation:

1. Start an OS/2 fullscreen session if one is not already available.
2. Change to the drive you want to install EASEL OS/2 EE on.

Note: Do not run the installation program from the A: prompt.

To start the installation, place Diskette 1 of the EASEL OS/2 EE Development System in drive A and enter

```
a:install
```

at the OS/2 command prompt.

During the installation process, you will be asked to provide the following information:

- The drive and directory into which EASEL OS/2 EE is to be installed (the default is C:\EASEL-EE).
- Whether you want the installation process to update the CONFIG.SYS file in the root directory of the fixed disk drive you boot your system from, or output the changes to a text file of your choosing (the default is CONF-NEW.ESL), which you must then manually edit into your CONFIG.SYS file.
- Whether or not you want to install support for audio capture and playback (the default is NO).
- Whether or not you want to install support for a touchscreen (the default is NO). If you do, you must modify the POINTER option in your CONFIG.ESL file to read:

```
POINTER=touchscreen
```

- Whether or not you want to change the standard codepage to be used by EASEL OS/2 EE programs (the default is 850, Multilingual).

After entering the drive and directory, the installation program checks for sufficient free space on the target fixed disk drive and if it is not available, generates an error. You are then asked if you want to quit. It is recom-

mended that you quit and remove files from the target fixed disk drive until at least 3.5MB of free space is available.

If your input is complete and correct and you choose to continue, you are prompted to insert additional diskettes when necessary.

When the procedure completes, you are prompted to remove the last diskette from drive A and you get a message telling you how to activate the changes made to the CONFIG.SYS file and prepare the system to use EASEL OS/2 EE. If you chose to have the installation process update the CONFIG.SYS file, the message tells you to reboot; if you chose to manually edit the CONFIG.SYS file, the message tells you to update the file and then reboot.

1.2.2 Installing Only the EASEL OS/2 EE Runtime Facility

Installation of the EASEL OS/2 EE Runtime Facility adds to your fixed disk drive only the applications, libraries, and files necessary for executing EASEL OS/2 EE graphical user interface applications.

Before you start the installation:

1. Start an OS/2 fullscreen session if one is not already available.
2. Change to the drive you want to install EASEL OS/2 EE on.

Note: Do not run the installation program from the A: prompt.

To start the installation, place Diskette 1 of the EASEL OS/2 EE Runtime Facility in drive A and enter

```
a:install
```

at the OS/2 command prompt.

During the installation process, you will be asked to provide the following information:

- The drive and directory into which EASEL OS/2 EE is to be loaded into (the default is C:\EASEL-EE).
- Whether you want the installation process to update the CONFIG.SYS file in the root directory of the fixed disk drive you boot your system from, or output the changes to a text file of your choosing (the default is CONF-NEW.ESL), which you must then manually edit into your CONFIG.SYS file.
- Whether or not you want to install support for audio capture and playback (the default is NO).
- Whether or not you want to install support for a touchscreen (the default is NO). If you do, you must modify the POINTER option in your CONFIG.ESL file to read:

```
POINTER=touchscreen
```

- Whether or not you want to change the standard codepage to be used by EASEL OS/2 EE programs (the default is 850, Multilingual).

After entering the drive and directory, the installation program checks for sufficient free space on the target fixed disk drive and if it is not available, generates an error. You are then asked if you want to quit. It is recommended that you quit and remove files from the target fixed disk drive until at least 1.5MB of free space is available.

If your input is complete and correct and you choose to continue, you are prompted to insert an additional diskette when necessary.

When the procedure completes, you are prompted to remove the last diskette from drive A and you get a message telling you how to activate the changes made to the CONFIG.SYS file and prepare the system to use EASEL OS/2 EE. If you chose to have the installation process update the CONFIG.SYS file, the message tells you to reboot; if you chose to manually edit the CONFIG.SYS file, the message tells you to update the file and then reboot.

1.2.3 Diskettes Required

You need the following diskettes for installation:

EASEL OS/2 EE Development System:

IBM EASEL for OS/2 Extended Edition
Development System Diskettes 1 and 2

IBM EASEL for OS/2 Extended Edition
Runtime Facility Diskettes 1 and 2

EASEL OS/2 EE Runtime Facility:

IBM EASEL for OS/2 Extended Edition
Runtime Facility Diskettes 1 and 2

These diskettes are available in two formats:

3.5" high density
5.25" high density

1.3 Directories Created and Files Added

1.3.1 Directories Created During Installation

The following directories are created during installation of the EASEL OS/2 EE Development System if you chose the default directory:

```
\EASEL-EE
\EASEL-EE\DLL
\EASEL-EE\MODULES
\EASEL-EE\PICTURES
```

Only the first two directories listed above are created if only the EASEL OS/2 EE Runtime Facility is installed.

1.3.2 Files Added During Installation

The following files are added to the target fixed disk drive during installation in the directories indicated, if you chose the default directory.

```
\EASEL-EE\
  A3270.EXE      Local application for 3270 communications
  A5250.EXE      Local application for 5250 communications
  ACCEL.INC      Include file for accelerator keys
  APPCPROF.EAL   Sample EASEL OS/2 EE APPC profile program
  COMPILE.CMD    Command file to initiate compiler
  CONFIG.ESL     EASEL OS/2 EE configuration file
  COPYRIGHT      EASEL OS/2 EE copyright information
  DATELIB.INC    Include file for the date library
  EASEL.CMD      Command file to initiate runtime
  EGACOLOR.INC   Include file for color palette functions (obsolete)
  EPARSE.EXE     EASEL OS/2 EE Compiler
  ESLAPPC.H      Declarations for APPC DLL support routines
  ESLAPPC.INC    Include file for APPC support
  ESLAPPC.LIB    APPC DLL support routines
  ESLAPPCA.EXE   Local application for APPC support
  ESLAPPCA.PRO   Example profile for APPC support
  ESLAPPCI.EXE   Local application for APPC support
  ESLAPPCI.PRO   Example profile for APPC support
  ESLMSG.EXE     OS/2 Edition error application
  ESLCUA.EXE     Layout/CUA tool
  ESLERR.TXT     EASEL OS/2 EE error message file
  ESSLIB.H       Declarations for EASEL OS/2 EE DLL support
                  routines
  ESSLIB.INC     Include file for the EASEL OS/2 EE library
```

ESLLIB.LIB	EASEL OS/2 EE DLL support routines
ESLPR.EXE	Text printing utility
ESLPRS.EXE	Text printing utility for the OS/2 spooler
ESLRUN.EXE	EASEL OS/2 EE Runtime
ESLSQL.BND	Bind file for SQL support
ESLSQL.EXE	Local application for SQL support
FIELDS.EXE	3270 fields utility
FIELDS52.EXE	5250 fields utility
FILEIO.INC	Include file for the file I/O library
MATHLIB.INC	Include file for the math library
MESSAGE.INC	Include file for the message library
PMPTR.INC	Include file for PM pointers
WELCOME.EAL	Sample EASEL OS/2 EE program
WELCOME.EBI	Sample EASEL OS/2 EE application

The following files are added if you selected the audio capture and playback option during installation:

AEDD_OS2.SYS	Audio support device driver
AUDIO.EBI	EASEL OS/2 EE audio utility
ESLAUDIO.INC	Include file for audio support
IBMAUDM.DSP	Audio support device driver
IBMAUDP.DSP	Audio support device driver
IBMAUDR.DSP	Audio support device driver
!ERASE.!AD	Audio support audio help file
!ERASE.!AU	Audio support audio help file
!PLAY.!AD	Audio support audio help file
!PLAY.!AU	Audio support audio help file
!QUALITY.!AD	Audio support audio help file
!QUALITY.!AU	Audio support audio help file
!RECORD.!AD	Audio support audio help file
!RECORD.!AU	Audio support audio help file
!SOURCE.!AD	Audio support audio help file
!SOURCE.!AU	Audio support audio help file
!START.!AD	Audio support audio help file
!START.!AU	Audio support audio help file

The following files are added if you selected the touchscreen option during installation:

TOUCHSCR.EXE	Touchscreen application
TSAPPL.EXE	Touchscreen calibration application
TSCAL.CMD	Command file to run touchscreen calibration
TSCALIB2.EBI	EASEL OS/2 EE touchscreen calibration application
TSECHO.EBI	EASEL OS/2 EE touchscreen calibration application

\EASEL-EE\DLL

DATELIB.DLL	Date library
ESLACDI.DLL	Asynchronous communications support library
ESLAPPC.DLL	APPC support library
ESLFONTS.FON	EASEL OS/2 EE fonts file
ESLLIB.DLL	EASEL OS/2 EE library
FILEIO.DLL	File I/O library
LOCAL.DLL	Local applications support library
MATHLIB.DLL	Math library
MESSAGE.DLL	Message library

The following file is added if you selected the audio capture and playback option during installation.

ESLAUDIO.DLL	Audio support library
--------------	-----------------------

\EASEL-EE\MODULES

BGRAPH.PRI	Business graphics module (private file)
BGRAPH.PUB	Business graphics module (public file)
CTEXT.PRI	Colored text module (private file)
CTEXT.PUB	Colored text module (public file)
M3270.PRI	3270 support module (private file)
M3270.PUB	3270 support module (public file)
M5250.PRI	5250 support module (private file)
M5250.PUB	5250 support module (public file)

\EASEL-EE\PICTURES

banking.inc	Banking Industry
bar.inc	Bar Graph
briefcas.inc	Briefcase
building.inc	Office Building
calendar.inc	Calendar
charge.inc	Charge Card
chemical.inc	Chemical Industry
clock.inc	Clock
compete.inc	Competition (2 horses)
confroom.inc	Conference Room
copier.inc	Copier
copyrite	Copyright notice
database.inc	Database
dish.inc	Satellite Dish
dollar.inc	Dollar Bill
dollsign.inc	Dollar Sign
drawer.inc	Drawer
eraser.inc	Eraser
factory.inc	Factory
filecab.inc	File Cabinet
finddoc.inc	Find Documents
folder.inc	File Folder

hide.inc	Hide
highpower.inc	Power Line
hiloclos.inc	Hi-Lo-Close Graph
hourglass.inc	Hourglass
ideabulb.inc	Light Bulb with Exclamation Point
key.inc	Key
linesym.inc	Line and Symbols Graph
lock.inc	Lock
mailbox.inc	Mailbox
mainfram.inc	Mainframe
notepad.inc	Notepad
office.inc	Office
offshore.inc	Oil Industry
openmail.inc	Open Mail
orgchart.inc	Organization Chart
otherdoc.inc	Other Documents
outmail.inc	Outgoing Mail
pen.inc	Pen
pencil.inc	Pencil
people.inc	3 People
pharmacy.inc	Pharmaceutical Industry
pie.inc	Pie Graph
plane.inc	Airplane
plug.inc	Electrical Industry
prepdoc.inc	Word Processing
printer.inc	Printer
proof.inc	Proofreading
questblb.inc	Light Bulb with Question Mark
question.inc	Question Mark
radioact.inc	Radioactivity
remind.inc	Reminder
samplebo.inc	Sample Book
scatter.inc	Scatter Graph
scissors.inc	Scissors
ship.inc	Ship
slide.inc	Slide
sprsheet.inc	Spreadsheet
stackbar.inc	Stacked Bar Graph
stckline.inc	Stacked Line Graph
stoplite.inc	Stoplight
telephon.inc	Telephone
terminal.inc	Computer Terminal
ticker.inc	Stock Ticker
ticket.inc	Airline Ticket
tools.inc	Tools

trashcan.inc
truck.inc
typewr.inc
usa.inc
van.inc
world.inc

Wastebasket
Truck
Typewriter
Outline of United States
Van
Globe

1.4 Testing the System

1.4.1 Testing the EASEL OS/2 EE Runtime

A sample application called WELCOME.EBI was added to the target fixed disk drive during installation. You can test the installation and configuration of the EASEL OS/2 EE Runtime by running this program. Simply change to the \EASEL-EE directory, or the one specified during installation if you did not choose the default, using the OS/2 CHDIR (or CD) command. Then, at the OS/2 command prompt, enter:

```
easel welcome
```

You can change the color of the window by selecting the Color choice from the action bar and then selecting one of the color choices from the Color pulldown. To exit the application, select the Exit choice from the Color pulldown.

1.4.2 Testing the EASEL OS/2 EE Compiler

If the EASEL OS/2 EE Development System was installed, a sample EASEL OS/2 EE program called WELCOME.EAL was added to the target fixed disk drive. You can test the installation and configuration of the EASEL OS/2 EE Compiler by compiling this program. Simply change to the \EASEL-EE directory, or the one specified during installation if you did not choose the default, using the OS/2 CHDIR (or CD) command. Then, at the OS/2 command prompt, enter:

```
compile welcome
```

Once the sample EASEL OS/2 EE program has compile, without any errors, you can execute it by following the instructions in the section above.

1.5 Installation Error Messages

The following error messages may appear during installation. The text enclosed in the < > symbols is replaced with the appropriate text for the particular error.

Group 1:

These error messages usually result from incorrect entries by the installer. The corrections are simple.

"ERROR - Invalid Directory Path."
"ERROR - Invalid FileName."
"ERROR - Invalid selection, try another."
"ERROR - Invalid value."
"ERROR - Path requires a drive specifier."
"ERROR - The filename can not contain path. Just a filename, please."
"ERROR - Path requires a backslash after the drive specifier."
"ERROR - Unable to make directory, <XXX> "
"ERROR - File already exists. Try another filename."

Group 2:

These error messages usually indicate a defective installation diskette or improper environment on the system on which EASEL OS/2 EE is being installed. They usually call for some other form of corrective action.

"ERROR - Invalid video mode, use TEXT mode."
"ERROR - Invalid video alpha format, use Text mode 80x25."
"ERROR - Invalid graphics video adapter."
"ERROR The Installation Data File, <XXX> , can not be found."
"ERROR Keyword, <XXX> , can not be found in the Installation Data File."
"ERROR Keyword Value for <XXX> , in the Installation Data File, is invalid."
"ERROR Can't open file, <XXX> "
"ERROR Can't create a temporary file"
"ERROR Can't create file, <XXX> "
"ERROR Deletion of file, <XXX> , failed."
"ERROR Rename of file, <XXX> , failed."
"ERROR The Installation Data File string exceeds max of 128."

"ERROR System call to command file failed."
"ERROR Serial Number file has an invalid format."
"ERROR Unable to update the OS2.INI file."
"ERROR Unable to Process CONFIG.SYS file. Contains a
line length greater than 254"
"ERROR Unable to revise PATH in CONFIG.SYS file. New
length would be greater than 254"
"ERROR Unable to revise LIBPATH in CONFIG.SYS file.
New length would be greater than 254"
"ERROR The LIBPATH in CONFIG.SYS is too long. It needs
to be shortened by < N > characters to create
FONTPATH."

Chapter 2. Welcome to EASEL OS/2 EE

2.1 Why EASEL OS/2 EE?

EASEL OS/2 EE, which is based on OS/2 Presentation Manager, is a graphical user interface development facility. It can support stand-alone applications, the workstation portion of cooperative processing applications, and applications that are front-ends to existing host terminal-based applications. The EASEL OS/2 EE portion of any application resides (and is developed) on an OS/2 Extended Edition programmable workstation (PWS). EASEL OS/2 EE runs on OS/2 Extended Edition, and supports the OS/2 Presentation Manager, OS/2 Extended Edition Communications Manager, and OS/2 Extended Edition Database Manager.

The EASEL OS/2 EE Layout/CUA tool provides developers a WYSIWYG (What You See Is What You Get) environment for creating user interfaces that conform to the IBM Systems Application Architecture™ (SAA™) Common User Access (CUA) guidelines. These CUA conforming user interfaces created with the Layout/CUA tool include a primary window, secondary windows, dialog boxes, and dialog controls to provide the user interface portion of an EASEL OS/2 EE application.

The EASEL OS/2 EE high-level language is event-driven, and is easy to learn. Language statements support graphics display, communications, and application logic. With the EASEL OS/2 EE language, developers can perform numerous tasks, such as:

- Modify the visual appearance of any object
- Read and write data files
- Re-format and display data graphically
- Interact with dynamic link libraries
- Access OS/2 relational databases
- Communicate with local and remote applications
- Communicate with host applications

In a stand-alone environment, EASEL OS/2 EE provides a means to easily and quickly develop graphical user interface applications that run on, and take advantage of, the OS/2 Presentation Manager. In a cooperative processing environment, EASEL OS/2 EE provides the capability to easily develop the workstation portion of a cooperative processing application, which runs in the OS/2 Presentation Manager environment. The partner

Systems Application Architecture and SAA are trademarks of International Business Machines Corporation.

portion of the cooperative processing application can be running either on a host or on another programmable workstation. In a host application font-ending environment, EASEL OS/2 EE provides a means for migrating existing terminal-based host applications to a CUA-style user interface running in the graphics-oriented OS/2 Presentation Manager environment.

2.1.1 Benefits of the EASEL OS/2 EE Language

The EASEL OS/2 EE language is English-like and straightforward in its syntax. Because EASEL OS/2 EE graphical user interface applications are so easy to develop, development time to create the application is significantly reduced. EASEL OS/2 EE provides support for:

- **High-resolution color graphics and text**

EASEL OS/2 EE is designed to support quality, detailed graphics and text—as simple as a row of rectangles representing selection choices, or as complex as a schematic layout of a factory floor. In business applications, a common use of graphics is to represent numeric data as line, bar, or pie graphs. Text can be mixed with graphics for the handling of textual material such as electronic mail.

- **Object-oriented-like program definition**

The images displayed on the screen are composed of named objects. The objects are related to each other via ancestry relationships and priority, and each object has attributes such as position, size, and color which determine its appearance on the screen.

- **Dynamic interaction with the user and external stimulus**

The objects displayed on the screen can be changed in response to any of the following events:

- Initiation of the EASEL OS/2 EE program
- An object on the screen being selected
- Text entered from the keyboard
- Stimulus received from a dynamic link library
- Text received from a local or remote application program
- Passage of time

By defining responses to various kinds of input from the user or from external stimulus, the programmer can change an object's color, size, position, or content, or cause other objects to appear, disappear, or change appearance.

EASEL OS/2 EE permits a flexible flow between different applications on different systems. The user is presented with a more convenient, less tedious flow of control than is possible with ordinary hierarchical menu systems.

- **Multiple methods of user input**

EASEL OS/2 EE supports the use of the keyboard as well as pointing devices such as the mouse and the touchscreen.

- **Communication with the host computer**

EASEL OS/2 EE is designed to communicate with existing host applications without requiring that it be modified, or communicate with a host in a cooperative processing environment. Communication is an integral part of EASEL OS/2 EE. An EASEL OS/2 EE program can perform for the user all the tasks necessary to dial up, log in, initiate programs, display information, log out, etc.

2.2 Writing an Event-Driven Application

With EASEL OS/2 EE, the developer creates an event-driven application. When running the application, EASEL OS/2 EE waits for an event, performs the actions that are defined in response to the event, and then waits for the next event.

2.2.1 EASEL OS/2 EE Front-End Application

An interaction between the user and a host application, using EASEL OS/2 EE as a front-end, might consist of the following actions.

Event	Response
1. The user starts the EASEL OS/2 EE program.	EASEL OS/2 EE displays the program's main window and action bar.
2. The user selects the action bar choice labeled "login...."	The program displays a login dialog box, which contains entry fields for "login name" and "password," and push buttons labeled "OK" and "Cancel."
3. The user enters login name and password, then selects the "OK" button.	The program logs onto the host system by sending it to the user's login name and password. If the login fails, the program displays a message in a message box asking the user to try again. If the login succeeds, the program makes the login dialog box invisible, sends a command to the host to start the host application, and enables the "Forecast" and "Logout" action bar choices.
4. The user selects the action bar choice labeled "Forecast."	The program sends a request for data to the host application. The resulting table is displayed in a window, and is used to create a graph, which is displayed in a different window.
5. The user selects the action bar choice labeled "Logout."	The program terminates the host application, then sends the logout command to the host. The program then exits.

The flow of this program is determined by events generated by the user. When the program is not performing a response to an event, it is idle.

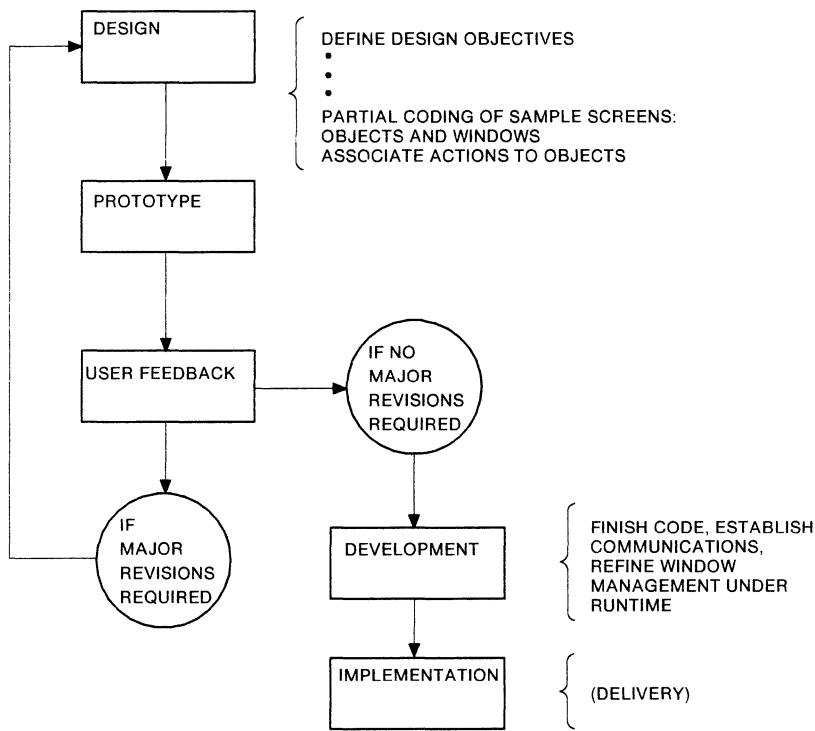
In order to build this interface program, the EASEL OS/2 EE developer would:

- Use the CUA screen layout tool to generate definitions and responses for the primary program window, action bar, and login dialog box.
- Add actions to the response defined by the CUA tool for the "Login" action bar choice to log in to the host computer and start the host application. These actions must recognize whether the login has been successful, and either display a message box indicating that the login has failed or proceed by making the login dialog box invisible and enabling the "Forecast" and "Logout" action bar choices.
- Define the response to the "Forecast" action bar choice. These actions must send the request for data to the host application, process the resulting data table, and display the results to the user.
- Define the response to the "Logout" action bar choice. These actions must exit from the host application, log out from the host, and then exit the program.

2.3 The Development Process

Below is the recommended approach to developing an EASEL OS/2 EE application.

THE DEVELOPMENT PROCESS



2.4 Distributing the Application

Once you have created a complete EASEL OS/2 EE application, you may want to distribute the executable EASEL OS/2 EE application (.EBI) file to a user community. Make sure the users' systems meet the EASEL OS/2 EE operating environment requirements, and provide them with the following files and information.

1. Indicate that each user must have the EASEL OS/2 EE Runtime Facility installed on their system. The EASEL OS/2 EE Runtime Facility must be the same version, or later, as the one used to create the EASEL OS/2 EE application. Otherwise, the application may not operate correctly.
2. Indicate which optional hardware is necessary to use the EASEL OS/2 EE application. Also ensure that the application is not limited to operating correctly with only one graphics display adapter resolution.
3. Provide instructions on how to install your EASEL OS/2 EE application, once the EASEL OS/2 EE Runtime Facility is installed, and how to invoke the application. You may want to provide an OS/2 command file (.CMD) which will automatically install the application, and another OS/2 command file to invoke the application.
4. Provide any dynamic link libraries and local applications necessary to use the EASEL OS/2 EE application which are not already provided with the EASEL OS/2 EE Runtime Facility. These should be installed along with the application.

Chapter 3. Programming with EASEL OS/2 EE

3.1 Summary of Changes

3.1.1 New Features

This version of EASEL OS/2 EE contains the following new features:

- **Additional dialog controls**
This enables an EASEL OS/2 EE application to utilize the additional dialog controls available in Presentation Manager. EASEL OS/2 EE provides the following new dialog controls:
 - **combination box**
 - **dropdown combination box**
 - **dropdown list**
 - **multiline entry field**
- **Dialog controls in windows (dialog region)**
This enables an EASEL OS/2 EE application to have primary and secondary windows which contain dialog controls.
- **SQL Support**
This enables an EASEL OS/2 EE application to access and manipulate OS/2 Extended Edition Database Manager relational databases.
- **Audio Support**
This enables an EASEL OS/2 EE application to provide audio playback and capture facilities as part of the user interface.
- **APPC Support**
This enables an EASEL OS/2 EE application to operate in a cooperative processing environment with a partner application running on a host or on another programmable workstation.
- **5250 Support**
This enables an EASEL OS/2 EE application to front-end host applications that use 5250 communications.
- **Response definitions to DLL events (response to stimulus)**
This enables an EASEL OS/2 EE application to respond to events generated by a dynamic link library.
- **Runtime command line variable initialization (-d)**
This enables EASEL OS/2 EE application variables to be initialized to particular values specified when the application is started.
- **Keyboard focus query (EslQueryFocus())**
This enables an EASEL OS/2 EE application to query the EASEL OS/2 EE object which currently has the keyboard focus.
- **PM window handle query (handle of)**
This enables an EASEL OS/2 EE application to query the PM window handle value of an EASEL OS/2 EE object.

- Internal EASEL OS/2 EE **subroutine** declaration
This enables EASEL OS/2 EE language statements to reference an EASEL OS/2 EE subroutine before it is defined.
- Fatal runtime error messages
This causes a message box to be displayed, before termination of the application, which informs the user that an EASEL OS/2 EE fatal runtime error has occurred.

3.1.2 Enhancements

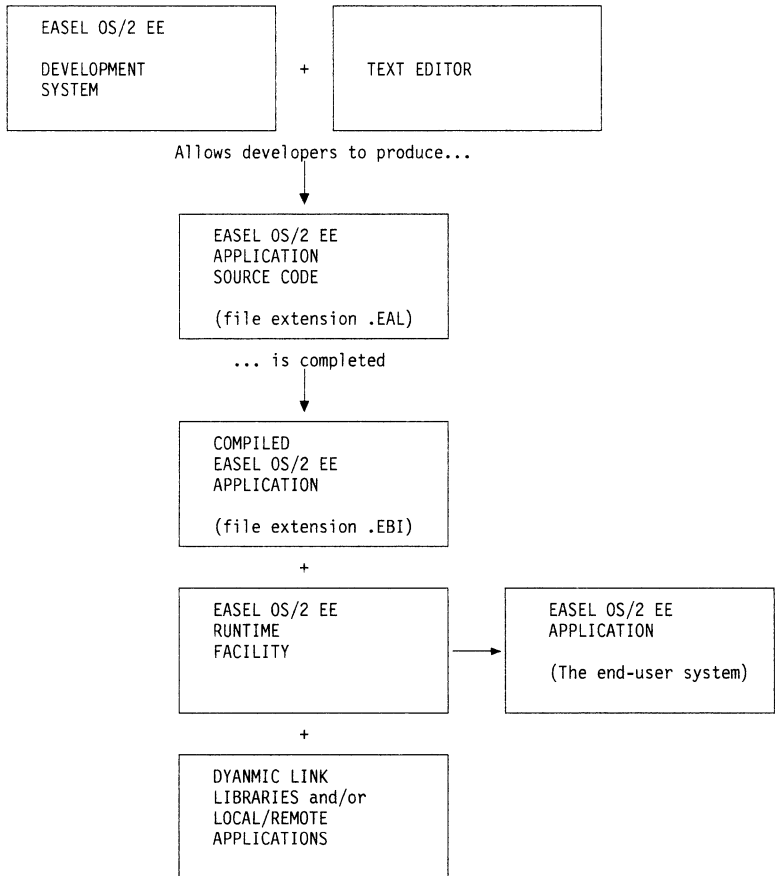
This version of EASEL OS/2 EE contains the following enhancements to existing features:

- Dialog controls
Enhancements now provide the following features:
 - Password entry fields
 - Horizontal scroll bars in list boxes
- Cascading pulldowns
Enhancements enable an EASEL OS/2 EE application to implement cascading pulldowns in a window's action bar.
- Layout/CUA
Enhancements provide the capability to include the new dialog controls, enhanced dialog controls, and cascading pulldowns in user interfaces created with Layout/CUA. Enhancements enable primary and secondary windows to contain dialog controls, and the region type of primary and secondary windows can be specified.
- 3270 Support
Enhancements enable an EASEL OS/2 EE application to control the appearance of the 3270 emulator window. New action routines enable an EASEL OS/2 EE application to switch to the 3270 emulator window and return as specified by the application.
- Response definition (**response to**)
Enhancements allow an EASEL OS/2 EE application to respond to the following user generated selections:
 - Activation of a window
 - Selection of the "Close" choice from the system menu
 - Double clicking with the mouse
- File I/O library
Modifications to the following subroutine names in the File I/O library avoid naming conflicts with EASEL OS/2 EE source code generated by Layout/CUA.
 - **Open()** is now **OpenFile()**
 - **Write()** is now **WriteString()**
 - **Close()** is now **CloseFile()**

Existing executable EASEL OS/2 EE applications (.EBIs) will continue to run correctly without any modifications. Although, any references to these subroutines in EASEL OS/2 EE source code will need to be updated with the new subroutine names before the program is re-compiled.

3.2 How an Application is Created

As shown in the following figure, you use the EASEL OS/2 EE Development System to create and compile source code for a graphical user interface application. The final application consists of compiled EASEL OS/2 EE code working with an EASEL OS/2 EE Runtime Facility and any dynamic link libraries (DLLs) or applications (local or remote).



3.3 Program Construction

An EASEL OS/2 EE program is a complete description of a set of objects and the responses to those objects. It contains all necessary definitions of the variables and constants used in the program and references to any dynamic link libraries or application programs with which the program communicates. Within a program you can display any number of screens of objects, making initially displayed objects invisible, and new objects visible.

3.3.1 Recommended Order of Program Components

An EASEL OS/2 EE program is composed of statements and definitions. Although there is no required order in which the statements and definitions that comprise your EASEL OS/2 EE program must appear—other than that every entity must be defined before it can be referenced—it is recommended that statements and definitions be ordered in such a way that they reflect the logical organization of the program.

The following lists the basic components of an EASEL OS/2 EE program, in the recommended order of appearance.

screen size Environment Declaration

Defines the coordinates of the screen.

font is Font Definition

Defines the font referenced as the program default font.

font Reference (Environment Declaration)

Specifies the default text font for the program.

precision Environment Declaration

Defines the precision used for floating point numbers.

module Reference(s)

References module(s) (sets of predefined EASEL OS/2 EE action statements and variables) to be used in the program.

application Environment Declaration(s)

Defines the logical name(s) for remote and/or local application programs with which the program will communicate.

stimulus Declaration(s)

Specifies the logical stimulus name(s) and its associated dynamic link library that implements that stimulus.

include Reference(s)

Specifies additional text file(s) of EASEL OS/2 EE code to be included in the program.

constant Definition(s)

Defines constant(s) used in the program.

variable Definition(s)

Defines variable(s) used in the program. You can define global variables, which can be referenced in all EASEL OS/2 EE programs that make up the application.

fileid Definition(s) and **record** Definition(s)

Defines information to be used by the file I/O library.

font is Font Definition(s)

Defines fonts used in the program.

pattern is Pattern Definition(s)

Defines a set of drawing statements.

imagemap Definition(s)

Defines a color mapping that can be used when reading in a bitmap image file.

action bar is Template Definition(s) and Item Definition(s)

Defines the action bar templates and items used within them.

Object Definitions

Defines the keys, graphical regions, image regions, textual regions, sense regions, dialog regions, dialog boxes, and dialog controls.

class Definition(s) and **item class** Definition(s)

Defines group(s) of objects or group(s) of items.

subroutine Declaration(s)

Declares subroutines defined later in the program.

action is Action Routine Definition(s) and **subroutine is** Definition(s)

Defines subset(s) of action statements to be referenced in response definitions (or in other **action is** Action Routine Definitions or **subroutine is** Definitions).

Response Definitions

Defines the event and the actions to be taken when the event occurs.

Note: Response Definitions always begin with a response statement and includes action statements.

3.3.2 Using Action Statements

Action Statements can only be specified in Response Definitions, **action is** Action Routine Definitions, and **subroutine is** Definitions. Action Statements are described throughout this publication, and are identified as such in the section titles and text.

3.4 Constructing a Program

3.4.1 An Introduction to Objects and Responses

EASEL OS/2 EE allows you to create, display, and manipulate objects. When you define an object, you specify its attributes. Some of the more common attributes are visibility, selectability, and position. You can also specify drawing statements, which define the appearance of the object. The drawing statements allow you to describe any shape, such as a box, a line, a star, and any text that an object will have.

The following is a sample object definition. Any text to the right of a pound sign (#) is a comment; the EASEL OS/2 EE processor ignores it.

```
primary region Primary size Width Height
  at position 100 100
  title bar "EASEL OS/2 EE"  #add a title bar
  system menu                #allow user to access
                              # system menu
  minimize button            #allow user to minimize
                              # region down to an icon
  maximize button            #allow user to enlarge
                              # region to full screen
  size border                #allow user to resize
                              # object

  move to (Width/2) (Height/2)
  text center "Welcome to EASEL OS/2 EE"
```

The EASEL OS/2 EE keyword **region** defines the object as a region, and the keyword **primary** defines this region as the main region. This region has a name (Primary), a rectangular size, a position at which it will be drawn, a title bar (used to move the object) with the text "EASEL OS/2 EE", a system menu key in the upper left corner, a minimize and maximize button in the upper right corner, and a size border. (See 4.3, "Keywords and Identifiers" on page 4-8 for a discussion of the rules for constructing names.) The drawing statements indicate that the text "Welcome to EASEL OS/2 EE" will be drawn centered in the middle of the region. In this example, Width and Height are constants.

When you define a response to an object, that response is performed if the object is **enabled** when the user selects the object displayed on the screen. The response definition contains:

- A **response** statement that specifies the event (or stimulus) that is to trigger the response. In the following example, the stimulus is the region *Primary* being selected.
- Action statements that describe the actions that EASEL OS/2 EE is to perform when the response statement is invoked.

The following is an example of a response definition:

```
response to Primary          # Response statement
  make Primary blue          # Action statements
  change Primary position by 10 10
```

This statement tells EASEL OS/2 EE to perform the action statements specified in the response definition only if the user selects the object named *Primary*. When selected, the color of the region is changed to blue, and the region changes position on the screen: the object disappears at the original position, and then reappears, in the color blue, at the new position.

3.4.2 Sample Program

The following sample EASEL OS/2 EE program builds on the previous sample by adding an action bar to the region and showing the definitions in the context of a complete, but simple, EASEL OS/2 EE program. The program demonstrates the following features:

- Constants
- Action bars
- Objects
- Responses
- Program termination

When the program begins, the primary region is displayed with an action bar that contains the choice *Color*. When the user selects the *Color* option, a pulldown menu, which gives the choices of *Red*, *Blue*, and *Exit*, is displayed. When the user selects *Red* or *Blue*, *Primary* is made that color. When the user selects *Exit*, the program exits.

```
##### Constants #####
# Once a constant has been defined, a reference to the constant is
# equivalent to a reference to its value. The use of constants makes
# the program easily modifiable. For example, if you decide to
# change the height of the primary region in this program, you can
# simply change the value of the constant Height, rather than
# changing the actual height specified in the size statement and also
# in the move to statement. The use of constants also helps to make
# your programs more readable and more self-documenting. For example,
# the meaning of the constant, Height, is more easily recognizable than
# value, 200.
integer constant Width is 300
                Height is 200
```

```
##### Action Bar Template Definition #####
# An action bar template is not a visible screen entity. It is only drawn
# on the screen when it is referenced by a region in its definition. The
# '-' character is the mnemonic for an action bar item. If the action
# bar is the active element in the program, then typing the character after
# the '-' is the same as clicking the mouse on that choice.
```

```
action bar Change is
    pulldown NewColor text "~Color"
    choice RedChoice text "~Red"
    choice BlueChoice text "~Blue"
    separator
    choice Exit text "E~xit"
end action bar
```

```
##### Object Definition #####
```

```
primary region Primary size Width Height
    at position 100 100
    title bar "Primary"
    system menu
    minimize button
    maximize button
    size border
    action bar Change      #      reference the action bar
    move to (Width/2) (Height/2)
    text center "Welcome to EASEL OS/2 EE"
```

```
##### Response Definitions #####
# Elements that compose action bars are referred to as items in #
# EASEL OS/2 EE. To create responses to items use the word item in #
# the response definition. #
#####
```

```
response to item RedChoice
```



```
make Primary red

response to item BlueChoice
make Primary blue

response to item Exit
exit
```

3.4.3 Trying Out EASEL OS/2 EE

Now that you have learned how to construct a simple EASEL OS/2 EE application, try compiling and executing the sample program *welcome.eal*. Follow the procedure described in 3.5.2, "Compiling an EASEL OS/2 EE OS/2 EE Application" on page 3-12 and 3.5.3, "Executing an EASEL OS/2 EE Application" on page 3-13.

- To change the color of the region, select the Color choice from the action bar and then select a color from the Color pulldown.
- To move the region, click and hold down the mouse button on the title bar and then reposition the region.
- To resize the region, click when the pointer is on the size border, hold down the mouse button, and move the region's edges to the desired size.
- To exit the program, select the Exit choice from the Color pulldown.

3.5 Creating, Compiling, and Executing an EASEL OS/2 EE Application

3.5.1 Creating an EASEL OS/2 EE Application

For each EASEL OS/2 EE application, you create an OS/2 text file containing EASEL OS/2 EE statements. The extension of this file must always be .EAL. A standard ASCII text editor can be used to create the .EAL files. Each .EAL file represents one application, although, an .EAL file can reference as many other files as necessary to be included as part of the EASEL OS/2 EE application.

To create the EASEL OS/2 EE application itself, an application developer must design and develop the user interface component of the application, and design and develop the application logic component of the application which performs the intended function.

For developing the user interface component of the application, a CUA user interface definition facility, called Layout/CUA is provided with EASEL OS/2 EE. Layout/CUA provides a What You See Is What You Get (WYSIWYG) environment for creating the primary and secondary windows and dialog boxes that make-up the user interface portion of the application. To use the Layout/CUA tool, enter the following at the OS/2 command prompt.

Model

eslcua

See Chapter 12, "Layout/CUA" on page 12-1 for information on using the Layout/CUA tool.

3.5.2 Compiling an EASEL OS/2 EE OS/2 EE Application

When an .EAL file is complete, you compile it using the following model at the OS/2 command prompt.

Model

compile PROGRAM_NAME [COMMAND_LINE_OPTION]...

In the model, `PROGRAM_NAME` is the OS/2 text filename *without* the `.EAL` extension, and `COMMAND_LINE_OPTION` is any options necessary for compiling the EASEL OS/2 EE application. These options are described in 22.2, “COMPILE.CMD OS/2 Command File” on page 22-3. Note that the **compile** command is controlled by the `COMPILE.CMD` file.

If errors occur during compilation, an executable EASEL OS/2 EE application will not be generated, otherwise a file with the same name as the source code filename, with an `.EBI` extension, will be generated.

3.5.3 Executing an EASEL OS/2 EE Application

Once an `.EBI` file is generated, you can execute it using the following model at the OS/2 command prompt.

Model

```
easel PROGRAM_NAME [COMMAND_LINE_OPTION]...
```

In the model, `PROGRAM_NAME` is the executable EASEL OS/2 EE application generated by the EASEL OS/2 EE Compiler, *without* the `.EBI` extension, and `COMMAND_LINE_OPTION` is any options necessary for executing the EASEL OS/2 EE application. These options are described in 22.4, “EASEL.CMD OS/2 Command File” on page 22-9. Note that the **easel** command is controlled by the `EASEL.CMD` file.

To see any errors that might occur during execution, specify **-e** `FILENAME` as a command line option, where `FILENAME` is the name of the error log file. Once the application has completed execution, the errors placed in the error log file can be viewed by entering type `FILENAME` at the OS/2 command prompt, where `FILENAME` is the same name specified with the **-e** command line option.

3.5.4 Editing the CONFIG.ESL File

The `CONFIG.ESL` file specifies configuration options that are used by the EASEL OS/2 EE Compiler and Runtime. This file can be updated as necessary, or a different configuration file can be specified for use with the Compiler or Runtime.

The statements in the CONFIG.ESL file, which is located in the directory specified during installation, must follow the following format:

Model

OPTION=VALUE

The values given in the file can be overridden by OS/2 environment settings. To set a value in the OS/2 environment, use the OS/2 SET command with the above model. For example, at the OS/2 command line prompt, enter:

set OPTION=VALUE

In the model, OPTION is the name of the configuration option you want to add or change, and VALUE is the new value you want to give the option. See 22.3, "CONFIG.ESL File" on page 22-5 for more information on setting specific values.

3.6 How EASEL OS/2 EE Compiles and Executes the Program

The EASEL OS/2 EE Development System parses an EASEL OS/2 EE program from top to bottom. Because of this hierarchy, you must define entities before you reference them. You must define an object before you can specify a response definition for it; you must define constants and variables before you use them in the initialization of other constants or variables, and you must define subroutines before you reference them in action statements.

If you have written a response for a class of objects and you have written a separate response for one of the objects that is in that class, the first response found—whether to the object as part of a class or to the object alone—is taken.

During runtime, if the program uses nested blocks, EASEL OS/2 EE begins with the current block. It checks the responses from the top of the block to the bottom; if no valid response is found, it continues in the same order in the next outer block. Thus, if you have written two responses for the same object, only the first response is executed.

Action statements within a response are executed in order, from top to bottom.

3.7 Frame Oriented Syntax

Regions can have any combination of the following PM frame attributes:

- Title bar
- Maximize and/or minimize buttons
- Horizontal and/or vertical scroll bars
- Size border or standard border
- System menu
- Action bar

You can use these frame attributes in all EASEL OS/2 EE region types, except sense regions. Some of these frame attributes can also be used in dialog boxes.

3.7.1 Screen Size

Specify the **screen size** environment declaration as the first statement in your EASEL OS/2 EE program, to define the screen coordinate system. The following describes the screen size environment definition as it applies to frames.

dialog units

Specifies that the screen coordinate system should be expressed in dialog units. Dialog units are proportional to the size of the PM system font. A dialog unit is defined to be one-quarter of the average character width of the system font in the horizontal direction, and one-eighth of the maximum baseline extent (that is, the maximum height) of the system font.

device units

Specifies that the screen coordinate system should correspond to the resolution of the display device. One coordinate in either dimension corresponds to one pixel on the display device.

Any object or drawing statement positioned outside of the specified coordinates will not be displayed.

Dialog units are useful for programs that make extensive use of the PM system font. Dialog boxes, dialog regions, and dialog controls are always defined in dialog units.

3.7.2 Defining Frame Related Attributes

The following describes the components of the region definition that apply to frames.

primary

Specifies that this region is the program's primary region.

title bar TITLE_STRING

Specifies that the region will contain a title bar containing the text specified by TITLE_STRING. This will generate a standard PM title bar that can be used for repositioning the region.

system menu

Specifies that the region will contain a system menu. The system menu button will be in the upper left corner.

horizontal scroll bar

Specifies that the region will contain a horizontal scroll bar along the bottom edge of the region, just inside the border, if one exists.

vertical scroll bar

Specifies that the region will contain a vertical scroll bar along the far right-hand edge of the region, just inside the border, if one exists.

no scale

Specifies that scaling of the region's contents should not occur when changing the region size (that is, the window size is constrained by the viewport). When the size of the region is changed, its window size is changed in parallel. This attribute is not used with textual regions or dialog regions.

action bar AB_NAME

Specifies that the region will contain the action bar specified in the template name AB_NAME.

minimize button

Specifies that the region will contain a minimize button in the upper right corner.

maximize button

Specifies that the region will contain a maximize button in the upper right corner. (The maximize button will turn into a restore button when the region is maximized.)

pointer [is] ...

Specifies the file containing the bitmap to use as a pointer.

The region size includes only the area of the region itself. It does not include the frame attributes, which are external to the size of the region. If the primary region is the same size as the screen, you cannot see any frame attributes.

3.7.3 Changing Title Bar Text

Use the **change text** action statement to change the displayed text of a region's title bar. You supply the identifier for a region containing a title bar and a string value representing the new text to be displayed in the title bar.

3.8 Windowed versus Fullscreen Applications

3.8.1 Overview

EASEL OS/2 EE allows you to create either a windowed or a fullscreen application.

- A windowed application is designed to share the screen with applications that are not controlled by the EASEL OS/2 EE application: word processing, mail, and so on. Windowed applications are best for users who are comfortable in a multi-tasking or windowed environment.
- A fullscreen application operates under Presentation Manager (PM), but once initiated, it takes control of the entire screen. Fullscreen applications are best for users who use their workstations for a specific task or set of tasks defined by the EASEL OS/2 EE application.

The default application type in EASEL OS/2 EE is windowed. A windowed application has one main region, which is defined with the attribute of **primary**. Actions performed on the primary region, such as minimizing it or deleting it, affect every other object in the EASEL OS/2 EE application, even objects that are not children of the primary object.

In a windowed application the primary region is also the default parent of any other object that is created (except for dialog boxes and dialog control objects). You can place any object except keys, sense regions, dialog boxes, and dialog controls outside the primary region by specifying **in desktop** for its ancestry, where desktop is defined as the overall display area. See 5.3.4, "Ancestry" on page 5-12 for a discussion of ancestry, parents, and children.

In order to create a fullscreen application, you must include the **-fullscreen** command line option when invoking the compiler (COMPILE.CMD). For fullscreen applications, there is no primary region. Instead, there is an object called **master**, which is the size of the screen and is the default parent of all other objects except dialog control objects.

3.8.2 Color Control

In a windowed application, all the PM applications that are running, including EASEL OS/2 EE, share the same color palette, which cannot be changed. The user can change the color of PM frame elements, such as the title bar, from the PM Control Panel.

In a fullscreen application, EASEL OS/2 EE allows the developer to change the second set of eight palette colors when running in EGA or VGA mode. Use the functions **EsiSetEgaColor()** and **EsiQueryEgaColor()** to set and query, respectively, the color palette. The colors you choose will not affect any other PM applications that are running.

3.8.3 Pointing Devices

A windowed application, because it is part of the PM environment, should use the mouse as the default pointing device. If the touchscreen is used in this environment, the user will not be able to access the other PM applications with the touchscreen. The user will also not be able to manipulate any PM frame attributes, such as action bars or dialog boxes, that are part of the EASEL OS/2 EE program. Therefore, when designing an application to use the touchscreen as the pointing device, you should not use PM frame attributes or dialog boxes as part of the EASEL OS/2 EE program.

A fullscreen application can have a mouse or touchscreen as a pointing device, as appropriate for particular users. However, if you design an application to use the touchscreen as the pointing device, do not use PM frame attributes or dialog boxes as part of the EASEL OS/2 EE program.

3.8.4 Scaling in a Windowed Application

In a windowed application, if size borders are used, the user can control the window size of that object via the PM controls without EASEL OS/2 EE having any knowledge of the change. When you create graphical regions and image regions with size borders, you can specify that when the object is resized, the contents of the object either:

- Scale, so that the same contents are displayed at different sizes.
- Clip without changing the scaling, so that more or less of the object is displayed.

The choice depends on the application. All other objects created with size borders clip the contents of the object as it is resized. Generally, objects that are mainly textual are better clipped because it is difficult to read text that is scaled very small, no matter what graphics display adapter you use.

3.8.5 Architectural Considerations in a Windowed Application

In fullscreen EASEL OS/2 EE applications the **change to program** syntax is often used to switch programs. In a windowed application, it is better to accomplish major program changes by having an EASEL OS/2 EE application invoke another EASEL OS/2 EE application. In this way, the window for the first application stays on the screen instead of disappearing when the new application's window is drawn (the user may be confused by seeing one window disappear and a new one appear in a different place).

3.8.6 Sizing Considerations

In general, you should design and initially display windows so that they are entirely visible on the screen. If windows are movable, users can move one off the screen so that some required controls are no longer accessible. You should do everything possible to provide users who do this with a way to continue.

The suggested method to accomplish this, is to provide the system menu icon with the window. This allows the user to select the "Move" choice from the system menu with the keyboard, by pressing Shift + Esc (or Alt + Spacebar) to access the system menu and then pressing the "M" key (or Alt + F1) to select the "Move" choice. The window can now be repositioned by the user with either the mouse or the keyboard.

3.8.7 System Menu Accelerators

In fullscreen applications, you cannot use the system menu accelerators to invoke the system menu choices. You can select the system menu choices with the mouse and mnemonics.

3.8.8 Examples

The following examples show how EASEL OS/2 EE works in a windowed application and in a fullscreen application.

3.8.8.1 Correctly Defined Region (Windowed)

The following is the right way to define a primary region in a windowed application:

```
# This will show up on the Desktop
```

```
screen size device units
enabled visible color 26 primary region PrimaryWindow
  size 600 325
  at 20 60
  size border
  color 27 foreground
```

```
system menu
title bar "PrimaryWindow"
no scale
action bar Primary_ActionBar
minimize button
maximize button
horizontal scroll bar
vertical scroll bar
```

3.8.8.2 Incorrectly Defined Region (Windowed)

The following is the wrong way to define a primary region in a windowed application, since the position of the window will be beyond the visible area on the display:

This will not show up on the Desktop

```
screen size device units
enabled visible color 26 primary region Primary Window
  size 600 325
  at 1000 1000
size border
color 27 foreground
system menu
title bar "Primary Window"
no scale
action bar Primary_ActionBar
minimize button
maximize button
horizontal scroll bar
vertical scroll bar
```

3.8.8.3 Correctly Defined Region (Fullscreen)

The following is the right way to define a region in a fullscreen application, since the position of the window will be beyond the visible area on the display:

This will show up on the screen

```
screen size device units
enabled visible color black region Main
  size 600 325
  at 20 60
```

3.8.8.4 Incorrectly Defined Region (Fullscreen)

The following is the wrong way to define a region in a fullscreen application, since the position of the window will be beyond the visible area on the display:

```
# This will not show up on the screen
```

```
screen size device units
enabled visible color black region Main
    size 600 325
    at 1000 1000
```

Chapter 4. Language Elements

4.1 The EASEL OS/2 EE Character Set

EASEL OS/2 EE supports the ASCII standard 7-bit character set. The EASEL OS/2 EE character set consists of the following ASCII characters:

Character	Meaning	Usage
SPACE	Blank	Throughout EASEL OS/2 EE
A-Z	Uppercase letters	Throughout EASEL OS/2 EE
a-z	Lowercase letters	Throughout EASEL OS/2 EE
0-9	Digits	Throughout EASEL OS/2 EE
=	Equal sign	Relational operator
+	Plus sign	Arithmetic operator
-	Minus sign	Arithmetic operator
*	Multiplication sign	Arithmetic operator
/	Division sign	Arithmetic operator
\	Backslash	EASEL OS/2 EE escape character
(	Left parenthesis	Required in expressions, subroutines, and functions
)	Right parenthesis	Required in expressions, subroutines, and functions
[	Left bracket	Required in arrays
]	Right bracket	Required in arrays
#	Pound sign	Comment character
,	Comma	Separator in argument lists and drawing statements
.	Decimal point	Floating point values
<	Less than	Relational operator
>	Greater than	Relational operator
"	Quotation mark	String delimiter
_	Underscore	Allowed in identifiers
:	Colon	Separator in argument lists and pattern scaling
All characters		String values

4.1.1 Blanks

EASEL OS/2 EE statements can appear in *free format*, which means that the specific format of characters on a line, or lines on a page, is not significant. For example, the following two statements are identical:

```
disabled key Hello at position 0 0
```

```
disabled key Hello
  at position 0 0
```


Wherever a space can appear in an EASEL OS/2 EE program, you can specify multiple spaces or tabs. The only exceptions to this rule are within string literals (see 4.4.3.4, "String Literals" on page 4-13) and comments (see below).

4.1.2 Comments

Comments help you, and anyone who examines the program in the future, to understand the logic of the program and the context and meaning of individual statements. Comments explain how statements work and why they are used. They are ignored by the EASEL OS/2 EE compiler.

Every comment must begin with a pound sign (#)—EASEL OS/2 EE comment character. All text from the comment character to the end of the line is part of the comment, even quotation marks and additional pound signs. Here is a full comment line:

```
# This program interfaces a "C" application.
```

A comment line cannot run over to the next line without another comment character. In the following example, the first line is a valid comment line, but the second line will be compiled as EASEL OS/2 EE code and will produce an error message:

```
# This program interfaces a  
C application.
```

The comment character need not appear at the beginning of a line. You can treat the right-hand portion of a line as a comment area. For example:

```
key Help at position 100 100      # See code for  
  box 50 50                      # "response to  
  move 10 10                     # Help"  
  text center "Help"
```

A comment character does not begin a comment when it is included in a quoted string. The following example does not contain a comment:

```
disabled key MessageA at position 100 150  
  " # Data Analysis Phase # "
```

4.2 National Language Support

4.2.1 Code Page Support

OS/2 uses code pages to change the way certain characters are displayed on the screen or printed on an output device. Each OS/2 code page defines 256 character codes. The first 128 characters are the standard ASCII character codes. The extended character codes include many international characters not found in the standard ASCII set. These five ASCII code pages are supported:

437	U.S.
850	Multilingual
860	Portuguese
863	Canadian-French
865	Nordic

Any language that EASEL OS/2 EE supports will work with the 850 (Multilingual) code page.

EASEL OS/2 EE allows you to compile and run your program using any of the five code pages; you can specify which code page to use. To activate code page support in EASEL OS/2 EE, the CODEPAGE, COUNTRY, and DEVINFO statements must be included in the OS/2 CONFIG.SYS file. For example, the following three statements in the CONFIG.SYS file specify the U.S. country code, the U.S. keyboard layout, and the Multilingual secondary code page, respectively.

```
COUNTRY=001,C:\OS2\SYSTEM\COUNTRY.SYS
DEVINFO=KEYBD,US,C:\OS2\KEYBOARD.DCP
CODEPAGE=437,850
```

These statements define your environment and which code pages will be available. EASEL OS/2 EE records and uses the currently active code page when compiling your program. This is also the code page used when the EASEL OS/2 EE program is run.

Because code page 850 supports any language that EASEL OS/2 EE supports, it is recommended that you set code page 850 as the primary active page in your CONFIG.SYS file and that you keep code page 850 as the active code page when you compile and run any EASEL OS/2 EE program.

If you want EASEL OS/2 EE to use a code page other than the currently loaded one, specify the new code page in the CONFIG.ESL file by adding the line:

```
CODEPAGE=NUM
```

where NUM is the number of the code page you want the program to be compiled and run with. See the *IBM Operating System/2 Extended Edition User's Guide* for the tables of code pages that show the character mappings to ASCII values.

Note: Remember to use the multilingual code page, 850, whenever possible. If you will be using 3270 communications in your program, you must compile and run your program with the same code page as the active OS/2 system code page. Using a code page other than the active system code page can cause characters to be displayed incorrectly.

If you are using 3270 communications in your program and you do not specify 850 as the current active code page, make sure that the CODEPAGE statement in the CONFIG.SYS file does not contain 850. Specify only one code page. For example, if you are using the Nordic code page 865, specify the following:

```
CODEPAGE=865
```

Because of the way the OS/2 Extended Edition Communications Manager (CM) is set up, if 850 is listed *anywhere* in the CONFIG.SYS CODEPAGE statement, an emulator will use code page 850. When you specify a code page other than 850 for communications, CM displays a warning screen before beginning the session, explaining that code page 850 is not being used. This screen is a reminder and does not indicate an error.

If you want to display the extended code page characters in your EASEL OS/2 EE program, use the PM font references instead of the standard EASEL OS/2 EE fonts. Standard EASEL OS/2 EE fonts do not have visual representations for the extended characters.

4.2.2 Keyboard Support

IBM OS/2 Extended Edition supports the following keyboards:

Country	Keyboard Code	Language
Belgium	BE	French
Denmark	DK	Danish
Netherlands	NL	Dutch
UK	UK166	English
UK	UK168	English
Finland	SU	Finnish
France	FR120	French

Country	Keyboard Code	Language
France	FR189	French
French Canada	CF	French
Germany	GR	German
Italy	IT141	Italian
Italy	IT142	Italian
Latin America	LA	Spanish
Norway	NO	Norwegian
Portugal	PO	Portuguese
Spain	SP	Spanish
Sweden	SV	Swedish
Switzerland	SF	French
Switzerland	SG	German
United States	US	English

The keyboard layout is defined to the OS/2 system in the CONFIG.SYS file using the DEVINFO=KBD statement. For example, the following statement would specify the Portuguese keyboard:

```
DEVINFO=KBD,P0,C:\OS2\KEYBOARD.DCP
```

See the *IBM Operating System/2 Extended Edition User's Guide* for more information on code pages.

For EASEL OS/2 EE applications, the keyboard must be specified in the CONFIG.SYS file. The OS/2 KEYB command will not change the keyboard for EASEL OS/2 EE applications.

If your keyboard layout is not the same as the one specified in the OS/2 system CONFIG.SYS file, you can still enter the extended characters from your keyboard by following these steps:

1. Look up the decimal character code associated with the symbol you want.
2. Hold down the left-hand <ALT> key and type the code on your workstation's numeric keypad.
3. Release the <ALT> key.

In any documentation accompanying a runtime application, be sure to specify the code page in which to run the application. You should also include in the documentation instructions for setting the code page and keyboard (if the application requires the use of a keyboard).

4.2.3 Date and Time Support

When developing applications that have National Language Support (NLS) considerations, the **date** and **time** Built-in Functions should not be used. Instead, the **LocalDate()** and **LocalTime()** functions should be specified as appropriate.

4.3 Keywords and Identifiers

EASEL OS/2 EE statement names, such as **variable**, **key**, and **response** are *keywords*, or reserved words. Attributes of objects, such as **color**, and words such as **position**, **size**, and **visible** are also keywords.

You must specify all EASEL OS/2 EE keywords in lowercase, exactly as they appear in this book.

An *identifier*, or user-defined name, is an arbitrary sequence of characters that begins with an uppercase letter. You specify a name only for entities that you define; these are:

- Action routines
- Applications (remote and local)
- Blocks
- Classes
- Constants
- File identifiers
- Fonts
- Image maps
- Loops
- Objects (keys, regions, dialog boxes, and dialog controls)
- Patterns
- Records
- Subroutines
- Variables

An identifier must follow these rules:

- It can contain any number of characters that fits on one line (255 characters).
- It must begin with an uppercase letter (A-Z) or an extended ASCII character. Because identifiers begin with an uppercase letter, EASEL OS/2 EE can distinguish the identifiers that you define from the keywords that are predefined in the EASEL OS/2 EE language.
- The remaining characters in the name can be uppercase or lowercase letters, digits, underscore characters (_), or an extended ASCII character.
- The mixture of case in which you specify an identifier is significant. The identifier *RedKey* is not the same as the identifier *Redkey*, and EASEL OS/2 EE treats these as two different things.

The following are valid identifiers:

B
RedKey
The_last_house_on_the_street

The following are invalid identifiers:

A* # WRONG: contains a special character
B 5 # WRONG: contains an embedded blank
fred # WRONG: does not begin with uppercase letter
5thStep # WRONG: does not begin with uppercase letter

It is a good practice to use easily readable, logical names that reflect the use of the entity. For example, the integer variable names “InnerLoopCounter” and “OuterLoopCounter” are more quickly understood than if they were named “I” and “J.”

Once an entity is defined, you can reference it simply by its name. If the entity is an object with ancestry, there are special rules for referencing it; see 5.3.4.2, “Specifying Ancestry in Object References” on page 5-13.

4.4 Values

4.4.1 Categories and Types of Values

Most EASEL OS/2 EE statements require you to specify one or more values. There are various categories of values, and every value has a type: **string** (text), **integer** (EASEL OS/2 EE integers), **float** (EASEL OS/2 EE floating point numbers), or **boolean** (**true** or **false**). Note that only constants and variables have default values.

Category	Types	Default	Examples
Literal	string		"status OK"
	integer		5
	float		27.96
	boolean		false
Constant	string	""	CompanyName
	integer	0	Counter_Max
	float	0.00	Pi
	boolean	false	Yes
Variable	string	""	Password
	integer	0	EmployeeNumber
	float	0.00	Grand_Total
	boolean	false	FirstTime
Variable preceded by a unary operator	integer		-A
	float		-Profit
	boolean		not AlreadyDone
Parenthesized Expression	string		(Message = "Final step")
	integer		(A * B + C)
	float		(A * 1.5)
	boolean		((A > B) and (C > D))
Built-in Function	string		members of Class2
	integer		xsize of GraphArea
	boolean		ioerror

Note: Built-in functions do not return floating point values.

In every EASEL OS/2 EE statement in which a value is required, you must specify a value of a particular type, or the default value will be used. The main rule to remember is that values must match the type. For example, when you specify coordinate positions in the definition of an object, integer values are required:

```
disabled key Message at position 50 100
```

```
key HelpKey at position Xpos Ypos
```

In the second example, Xpos and Ypos are constants or variables that contain integer values.

When you specify data to be sent to an application program, a string value is required:

```
send "Logoff\n" to Newswire
```

When you specify a loop, a boolean value is required:

```
while (true) loop
```

or

```
while (A > B) loop
```

```
# If A is greater than B, the loop will be executed.
```

4.4.2 Compile-time Values

A compile-time value is a value that is evaluated at the time the program is compiled. (These values are "compiled into" the EASEL OS/2 EE program at compile time.) A compile-time value should be a literal, a constant, or an expression containing only literals and/or constants as operands. They can be (or contain) variables, but the initial variable's compile time value will be used.

Examples of statements that require compile-time values are:

```
screen size (Size + Size/3) (Size)
```

```
# The screen size requires two
```

```
# compile-time values; Size is a constant.
```

```
font "medium"
```

```
# Specifies default font for the program.
```

```
response to line "Login" from Host
```

```
# The match string for a response must be a compile-time
```

```
# value.
```

4.4.3 Literals

4.4.3.1 Integer Literals

The range of valid numbers in an integer literal is -2^{32} to $2^{32}-1$. When you specify an integer literal, you can specify minus signs and leading zeros. The following are valid integer literals:

```
0
-5
010
-9999
```

All other characters are invalid, including embedded blanks, commas, and decimals. The following are not valid integer literals:

```
3,942,885    # WRONG: contains commas
5E7          # WRONG: contains an alphabetic character
1.23         # WRONG: contains a decimal
2 3/4        # WRONG: contains an embedded blank and
              # a division sign
```

An integer literal can be specified in any context in which an integer value is required; for example:

```
integer variable Max is 100000    # integer variable

key Key1 at position 50 75        # graphical coordinates

move to column 2 line 7           # textual region
                                  # coordinates

copy (Counter +1) to Counter      # arithmetic expression
```

4.4.3.2 Floating Point Literals

The range of valid numbers for a floating point literal is 16 places. This is the total number of places, on both sides of the decimal point. EASEL OS/2 EE ignores numbers beyond the 16th place. Double precision is not allowed.

When you specify a floating point literal, you can specify minus signs and leading zeros in front of the decimal point. The following are valid floating point literals:

```
0.01
-.5      # The leading zero is understood.
.0103
-9999.01
```

All other characters are invalid, including scientific notations, embedded blanks, and commas. The following are not valid floating point literals:

```
3,942,885.03      # WRONG: contains commas
.05E7             # WRONG: contains an alphabetic character
.02 3             # WRONG: contains an embedded blank
```

A floating point literal can be specified in any context in which a floating point value is required; for example:

```
float constant Rate is .9.895
```

4.4.3.3 Changing the Precision

You can change the precision with which floating point values are output by using the **precision** environment declaration. For example:

```
precision 3
float Max is 23.0
```

```
disabled key DisplayMax at position 30 40
text Max
```

will produce the following text in the disabled key DisplayMax:

```
23.000
```

If no precision is specified, the default value of **2** is used (which truncates the value after two digits to the right of the decimal point). If a precision of 0 is specified, the decimal point will be displayed, with no digits to its right.

4.4.3.4 String Literals

A string literal is a text string consisting of zero or more ASCII characters enclosed in quotation marks (""). The following are valid string literals:

```
""           # Length zero; called a null string.
" "          # A single blank.
"A"
"2"
"Logging On: Please Wait"
"Select MORE to continue"
```

The following are not valid string literals:

```
"Begin here      # WRONG: Not ended by quotation mark.
'OK to continue' # WRONG: Delimited by apostrophes, not
                  # quotation marks.
```

A string literal can be specified in any context in which a string value is required; for example:

```
string variable Message is "Select anywhere to start"
```

```
disabled key ColorPrompt at position 50 50
    text "Press the desired color key"
```

```
send "Run DTR\n" to Newswire
```

The compiler supports a maximum string literal length of 255 characters.

4.4.3.5 Using Escape Sequences in String Literals

In a string literal, you can specify special control characters and characters that might not be available on your keyboard, such as new lines (line feeds) and carriage returns. If you specify such characters in a string, you must use an *escape sequence*. This consists of the EASEL OS/2 EE escape character—a backslash (\)—followed by a code representing the special character. The escape character itself is not part of the string; it tells EASEL OS/2 EE that you want to “escape” from the string to specify a character that cannot be specified explicitly in the string.

4.4.3.6 Escape Sequences Table

The following are escape sequences supported by EASEL OS/2 EE:

Escape Sequence	Meaning
\"	Quotation mark
\\	Backslash
\a	Right align in action bars
\b (or \^H)	Backspace (destructive)
\e (or \^[)	Escape
\f (or \^L)	Form feed (new page)
\n (or \^J)	Line feed (new line)
\r (or \^M)	Carriage return
\t (or \^I)	Tab (horizontal)
\v (or \^K)	Tab (vertical)

Note: The single character escape sequence `\e` (or `\^[`) should be used to specify the ASCII Escape character in a string literal whereas the double character keyboard character code `\e\e` (or `\^[\[`) should be used to respond to the Escape (Esc) key typed on the keyboard (see 15.3.1.1, “Keyboard Mapping” on page 15-9).

When you send data to an application program, you often must end the data with the new-line escape sequence. This sequence simulates the RETURN key on the keyboard. Most applications do not recognize input until they receive the new-line character. To specify a new-line character in a string literal, you specify the EASEL OS/2 EE escape character (\) followed by a lowercase **n** or a **^J**, which represents a new-line character:

```
"OK\n"
```

or

```
"OK^J"
```

The length of either of these strings is three characters; "OK" followed by a new-line character.

You cannot specify a quotation mark directly within a string literal because EASEL OS/2 EE recognizes the second quotation mark as the end of the literal. However, by specifying an escape character before the quotation mark, you tell EASEL OS/2 EE to escape from the string and process the character. For example, to display the following string on the screen:

Type "OK" when you are ready

you specify the following:

```
"Type \"OK\" when you are ready"
```

4.4.3.7 Boolean Literals

A boolean literal is a boolean value: **true** or **false**.

The following are not valid boolean literals:

```
True      # WRONG: Not lowercase.
"true"    # WRONG: These are string literals, not
"false"   # boolean literals.
```

A boolean literal can be specified in any context in which a boolean value is required; for example:

```
response to BeginL
  while (true) loop
    begin
      response to Done
    :
      leave loop
  end
end loop
```

4.4.4 Constants

The use of variables and constants, in place of literals, helps make your programs more general, more readable, and more easily modified. You define as variables those entities that will change value during execution. You define as constants those entities that will retain their initial values throughout the program.

A constant is a named entity that holds a data value. You define the names, types, and values of constants, and then reference them in your program. The value specified in the constant's definition is permanent; it cannot be changed during execution.

You can use constants to identify values required by EASEL OS/2 EE statements. By specifying constant names rather than literals, you make it easier to change these values at some future time (before recompiling).

Suppose you define constants representing the width of a key and the vertical placement of a row of keys. After writing your EASEL OS/2 EE program, you might decide that the appearance of the graphics would be improved by changing the width and/or the placement of the keys. If you have defined the width and placement as constants, rather than literals, you will be able to change these values by simply changing the constant statement, rather than by modifying every **key** statement in the program.

4.4.4.1 Defining Constants

The **constant** statement defines one or more constants. A constant must be defined before it is referenced. When you define a constant, you define its type by specifying the keyword **integer**, **float**, **string**, or **boolean**. If no type is specified, the constant is a **string** by default.

Model

```
[ integer | float | string | boolean ] constant  
  CON1 is VALUE1  
  [CON2 is VALUE2]...
```

For example:

```
integer constant Max_Retries is 3 Max_Timeout is 5
```

Regardless of the constant type, you cannot omit the keyword **constant**. If you do, you will be defining a variable, not a constant.

The type that you specify for a constant determines the type of value required for the initialization. Wherever an integer or floating point value is

expected, you can use a negative value, such as -3 or -4.03. Wherever a boolean value is expected, you can specify a negative boolean, such as:

```
boolean constant SystemOff is  
    (not SystemOn)
```

where SystemOn is a boolean value.

4.4.5 Variables

Like constants, a variable is a named entity that holds a data value. You can define the names, types, and initial values of the variables that you will use in your programs. Unlike constants, however, the values of variables can be changed during execution.

4.4.5.1 Defining Variables

The **variable** statement defines one or more variables. A variable must be defined before it is referenced. When you define a variable, you define its type by specifying the keyword **integer**, **float**, **string**, or **boolean**. If no type is specified, **string** is the default.

Model

```
[global] { integer [variable] | float [variable] |  
           boolean [variable] | string [variable] |  
           [string] variable }  
VAR1 [is VALUE1]  
[VAR2 [is VALUE2] ]...
```

For example:

```
integer variable Index
```

If you specify a variable type, you can omit the keyword **variable**, such as:

```
boolean MainframeIsDown      # defaults to variable
```

If you omit the type, however, you must specify the keyword **variable**; for example:

```
variable Counter              # defaults to string
```

4.4.5.2 Initializing Variables

You can assign an initial value to the variable when you define it, such as:

```
integer variable Index is 1  
integer variable Box_XCenter is (BoxWidth/2)
```

The type that you specify for a variable determines the type of value required for the initialization (if any). Wherever an integer or floating point value is expected, you can use a negative value, such as -2 or -43.21. Wherever a boolean value is expected, you can specify a negative boolean, such as:

```
boolean variable Using_Dummy_Data is (not LoggedOn)
```

4.4.5.3 Global and Local Variables

If you are using multiple EASEL OS/2 EE programs in your EASEL OS/2 EE interface, you can define and reference variables in all the EASEL OS/2 EE programs to be executed. These are called **global** variables and are defined as such; for example:

```
global variable MessageCount
```

Suppose that two programs each define a global variable named MessageCount. When one of the programs transfers to the other program, the value of MessageCount is retained and is available for use by the new program.

All programs that use the same global variable must define it as a global variable, specifying the same name and the same type. If the new program initializes the global variable, the initialization does not override the value left behind by the earlier program. Ordinarily, initialization is specified only in the first program.

If the variable is not defined as global, the variable defaults to a local variable. For local variables, values assigned will be retained only while the current program is executing. A local variable ceases to exist when EASEL OS/2 EE leaves the program in which the variable is defined. If the program is re-entered, the value is reinitialized.

If you want to retain the value of a variable between invocations when changing between many EASEL OS/2 EE programs, you must define the variable as a global variable, even if none of the other programs use that variable.

You can save the current value of the local variables in a program by saving the program with a **save program as** statement. See 11.1.5, "Global and Local Entities" on page 11-9 for more information on how global variables work among different EASEL OS/2 EE programs.

4.4.5.4 Finding the Length of a String Variable

You can find the number of characters in any string variable during execution by using the **length of** built-in function. For example, if you have defined a string variable as follows:

string Message is "Select to log on"

and then specify later in the program the function:

length of Message

EASEL OS/2 EE returns the integer value 16. Note that spaces are counted.

4.5 Assigning Values to Variables During Runtime

You can assign new values to variables during execution by specifying **copy**, **append**, or **extract from** action statements in your EASEL OS/2 EE program. (Remember that you cannot modify constants.)

4.5.1 The copy Action Statement

The **copy** action statement assigns a new value to a variable. Any existing value of the variable to which you are copying is replaced by the new value.

Model

```
copy VALUE1 [VALUE2]... to { clipboard | VAR | VAR[ N1[, N2]... ] }
```

For example:

```
# The contents of the variable Widgets are copied into
# the variable named Inventory:

response to UpdateInventory
  copy Widgets to Inventory

# The boolean literal false is copied into the variable
# LoggedOn:

response to timeout 10
  copy false to LoggedOn

# The floating point value 1473.69 is copied into the
# floating point variable TotalSales:

response to start
  copy 1473.69 to TotalSales

# The result of the expression (A * 5 - B) is copied into
# the variable Calc:

response to ...
  copy (A * 5 - B) to Calc

# Copying the null string ("") into the variable Homework
# deletes the existing contents of Homework and replaces
# it with the null string.
```

```
response to ...
  copy "" to Homework
```

In the following example, the string literal "OK" and the value of the string variable FirstStatus are copied into the variable StatusMessage. When more than one value is copied to a variable, the values are concatenated. If FirstStatus contains the string "ChecksOut", the value of StatusMessage after the **copy** statement is executed is "OKChecksOut".

```
response to Acknowledge
  copy "OK" FirstStatus to StatusMessage
```

You can also copy selected text from a textual region to a variable using the **copy** statement; see 8.5.2, "Using the textual Built-in Functions" on page 8-42.

When you copy a value to a variable, the type of variable determines the type of value that EASEL OS/2 EE expects. If the new value is of a different type, EASEL OS/2 EE will try to convert the value to the expected type.

4.5.2 The append Action Statement

The **append** action statement appends one or more values to the existing contents of a string variable. You cannot append values to an array element.

Model

```
append VALUE1 [VALUE2]... to STRING_VAR_NAME
```

For example:

```
response to ...
  append ".05" to Dollars # The string ".05" is appended
                        # to the variable Dollars.
```

In the following example, Citylist contains the names of cities, each followed by a blank ("N.Y. L.A. Chicago "); the variable MA contains two city names ("Boston Springfield "); and the variable FL contains a single city name ("Miami ").

```
response to ...
  append MA FL to Citylist
```

After EASEL OS/2 EE executes the **append** statement, the contents of the variables MA and FL are appended to the existing contents of the variable named Citylist. The new value of Citylist is:

```
"N.Y. L.A. Chicago Boston Springfield Miami "
```

You can append selected text from a textual region to a variable using the **append** statement.

4.5.3 The **extract from** Action Statement

The **extract from** action statement extracts data fields from a string value and copies those fields to one or more specified variables. It is used to parse through a string, breaking it down into other variable values; most often, it is used to examine and store the input that is received from an application or from the keyboard. Nevertheless, you can use it to extract data fields from any string value. Because this statement is so versatile, a fairly lengthy description follows. You may wish to familiarize yourself with the EASEL OS/2 EE coordinate systems, textual regions, built-in functions, and communications before studying this section.

Model

extract from SOURCE

```
[SKIP_CLAUSE]...  
TAKE_CLAUSE  
[ TAKE_CLAUSE | SKIP_CLAUSE ]...
```

In the **extract from** statement, you specify a **SOURCE**, which is the source of the input being examined—the string from which you want to extract data fields. You also tell EASEL OS/2 EE how to locate the desired data fields by specifying a series of **TAKE** and **SKIP** clauses. EASEL OS/2 EE processes the source, usually from left to right, applying the **TAKE** and **SKIP** clauses in the order in which the clauses are specified. The models for these clauses are shown below.

The **TAKE** clauses tell EASEL OS/2 EE to take particular data fields in the **SOURCE** as the values of the specified variables. The **SKIP** clauses tell EASEL OS/2 EE to skip over data fields in the **SOURCE**. You must specify at least one **TAKE** clause in an **extract from** statement. You can specify as many additional **TAKE** and **SKIP** clauses as necessary to process the source. The **TAKE** and **SKIP** clauses must follow immediately after the **extract from** statement, which cannot contain any other action statements. The string value for the **SOURCE** entry can be the name of a variable containing a string value, or a built-in function such as **parameter** or **input**. For example, you can specify:

```
extract from textual line N from TextRegion3  
:
```

or:

extract from parameter of Help_Key
:

TAKE_CLAUSE follows the model:

Model

```
take [ by | to ] { marker | last | word | number |  
                  float | TEXT | COLUMN } VAR_NAME
```

SKIP_CLAUSE follows the model:

Model

```
skip [ by | to ] { marker | last | word | number |  
                  float | TEXT | COLUMN }
```

In both clauses, COLUMN is an integer value representing a column in the SOURCE string; TEXT is a string value representing the text being extracted or skipped. In the TAKE clause, VAR_NAME is the identifier for the variable to which the extracted string will be copied.

As the source is processed, an internal pointer keeps track of the current position in the source caused by the SKIP and TAKE movements. At the beginning of an **extract from** statement, the pointer is at position 1 (the first character, or *column*, in the source). As EASEL OS/2 EE executes the TAKE and SKIP clauses, it moves this pointer. The only difference between the TAKE and SKIP clauses is that TAKE both moves the pointer and copies the processed characters into the variable, whereas SKIP simply moves the pointer. The value of the pointer can be accessed through the **marker** built-in function.

An example of a response definition containing a complete **extract from** statement, with several TAKE and SKIP clauses, is shown below.

```
response to line from Application2  
  extract from input  
    take word KeyName      # Name of key to move  
    skip "X="              #  
    take number NewX       # New X coordinate  
    skip "Y="              #  
    take number NewY       # New Y coordinate  
  change KeyName position to NewX NewY
```

In this example, the response is invoked when a line of text is received from an application program. (The **input** function is used to ask for the string that stimulated the current response.) If the line contains the string:

A34 X=53, Y=106

then this example extracts three fields from the string returned by the **input** function, and assigns these values to the variables **KeyName**, **NewX**, and **NewY**. The statement contains three **TAKE** clauses and two **SKIP** clauses. These clauses have the following purposes:

take Specify the data fields that are to be extracted and stored as the values of the variables **KeyName**, **NewX**, and **NewY**.

skip Specify the data fields that are to be skipped.

In processing the source, EASEL OS/2 EE examines characters from left to right and does the following:

1. Assigns the first word in the source ("A34") as the string value of the variable **KeyName**.
2. Skips the field, "X= ".
3. Copies the number after "X= " (53) to the integer variable **NewX**.
4. Finds and skips the field "Y= ".
5. Copies the number after "Y= " (106) to the integer variable **NewY**.

After the **extract from** statement has been executed, the statement:

change **KeyName** to position **NewX NewY**

moves the key named **A34** to the **X Y** coordinates **53 106**. Note, also, that in all **TAKE** and **SKIP** statements, the keyword **by** is optional; it specifies *relative* movement of the pointer through the source string. The keyword **to** specifies *absolute* pointer movement.

The rest of this section describes how each of the keywords and identifiers (**marker**, **last**, **word**, **number**, **float**, **TEXT**, and **COLUMN**) are used in the **SKIP** and **TAKE** clauses.

4.5.3.1 take to or skip to marker

After EASEL OS/2 EE has executed an **extract from** statement, the pointer's position is recorded in the built-in function **marker**. This function is useful when processing a number of **extract from** statements on the same input string. Consider the following example. (Note that many of the following examples contain **SKIP** and **TAKE** clauses that are addressed in later sections; this is because **extract from** statements typically contain various types of **SKIP** and **TAKE** clauses. You need not try to figure out every clause in each example; rather, focus on those clauses that are being addressed.)

```

extract from TitleLine
  take to "*" GTitle
extract from ""
  take 1 Reset      # Sets pointer (marker) value to 1
while (marker <= length of input) loop
  extract from input
    skip to marker
    take number X
    take number Y
  add to Graph
  draw to X Y
  copy (1 + GraphPoint) to GraphPoint
end loop

```

In this example, the variable TitleLine contains the following; the pointer is initially at column 1, as shown:

Q2 Sales * (from file C:\temp)

```

-----
^

```

Throughout these examples, the dashes show the actual space counts of the source, and the caret (^) shows the position of the pointer.

When EASEL OS/2 EE executes:

```
take to "*" GTitle
```

the string "Q2 Sales" is taken and copied to the variable GTitle. The pointer is at column 9, as shown:

Q2 Sales * (from file C:\temp)

```

-----
      ^

```

Then, when EASEL OS/2 EE executes:

```

extract from ""
  take 1 Reset

```

the pointer returns to column 1, and the value of **marker** is set to 1. Now EASEL OS/2 EE executes the **extract from input** statement (with the pointer at column 1); in this example, **input** contains the following:

10 25 15 37 20 31

```

-----
^

```

When EASEL OS/2 EE executes:

```

take number X
take number Y

```

the number "10" is taken and copied to X, and the number "25" is taken and copied to Y. The pointer moves as shown below, and the value of **marker** changes to 6:

```
10 25 15 37 20 31
-----
      ^
```

Then, when EASEL OS/2 EE executes:

```
skip to marker
  take number X
  take number Y
```

the number "15" is taken and copied to X, and the number "37" is taken and copied to Y. The pointer moves as shown below, and the value of **marker** changes to 12.

```
10 25 15 37 20 31
-----
      ^
```

4.5.3.2 take [by] or skip [by] marker

These clauses are not meaningful.

4.5.3.3 take to or skip to last

The **take to last** or **skip to last** statement starts with the current pointer position and skips or takes all subsequent characters up to and including the last character in the source. The pointer is then positioned after the last character of the source. Consider the following example:

```
extract from MessageLog
  take number MessageNumber
  take word Person
  skip "DIAGNOSTIC:"
  take to last MessageText
```

MessageLog contains the following:

```
  3  FRED  DIAGNOSTIC: BAD VALUE
-----
      ^
```

At the beginning of an **extract from** statement, the pointer is always at position 1. In this example, the following statement is executed first:

```
take number MessageNumber
```


EASEL OS/2 EE skips over the leading blanks and takes "3" as the value of the variable MessageNumber. The pointer is now positioned on the space after the "3". Next, EASEL OS/2 EE executes:

```
take word Person
```

EASEL OS/2 EE skips over the leading blanks and takes "Fred" as the value of the variable Person. The pointer is now positioned on the space after the "D". Next, EASEL OS/2 EE executes:

```
skip "DIAGNOSTIC:"
```

EASEL OS/2 EE skips over the specified string and leaves the pointer on the space after the colon, as shown below.

```
3 FRED DIAGNOSTIC: BAD VALUE
-----
                        ^
```

Finally, EASEL OS/2 EE executes:

```
take to last MessageText
```

EASEL OS/2 EE starts at the current pointer position and takes all characters through the last character as the value of the variable MessageText. The variable has the value " BAD VALUE:". The pointer is now positioned, as follows, after the end of the string.

```
3 FRED DIAGNOSTIC: BAD VALUE
-----
                        ^
```

4.5.3.4 take [by] or skip[by] last

These clauses are not meaningful; an error message will be produced.

4.5.3.5 take [by] or skip [by] word

Either of these statements starts with the current pointer position, skips over any leading blank characters, and then skips or takes all subsequent characters up to, but not including, the first nonprinting character (e.g., blank) or the end of the source. (Numeric characters are considered part of the word.) After executing either of these statements, the pointer is positioned immediately after the end of the skipped or taken word.

Consider the following example:

```
extract from StringA
  take word Var1
  skip word
  take word Var2
```

StringA contains the following. Note that the pointer is initially positioned at column 1.

```
CITY IS BOSTON
-----
^
```

In this example, the following statement is executed first:

```
take word Var1
```

EASEL OS/2 EE skips over the leading blanks. The word "CITY" is taken and copied to the variable Var1. The pointer moves as shown:

```
CITY IS BOSTON
-----
  ^
```

Next, EASEL OS/2 EE executes:

```
skip word
```

The word "IS" is skipped. The pointer moves as shown:

```
CITY IS BOSTON
-----
    ^
```

Finally, when EASEL OS/2 EE executes:

```
take word Var2
```

the word "BOSTON" is taken and copied to the variable Var2. EASEL OS/2 EE skips over the leading blank. The pointer moves as shown below.

```
CITY IS BOSTON
-----
        ^
```

If EASEL OS/2 EE cannot find a word, the pointer does not move, and, in TAKE clauses, the null string is copied to the specified variable. This might occur if the pointer is at the end of the source, or if there are only blanks between the pointer and the end of the source.

4.5.3.6 take to or skip to word

Either of these statements starts with the current pointer position and skips or takes subsequent characters up to, but not including, the first printing character encountered. After execution, the pointer is positioned at the beginning of the word. If EASEL OS/2 EE cannot find a word, the pointer does not move, and, in TAKE clauses, the null string is copied to the specified variable.

4.5.3.7 take [by] or skip [by] number

Either of these statements starts with the current pointer position, skips over any leading blanks, and then skips or takes subsequent digits (which may include a leading sign), up to the first nondigit (e.g., a blank, a decimal point, or an alphabetic character) or the end of the source.

Note: These clauses do not pass over alphabetic characters. For example, if the source contains the following, and the pointer is on the first character (A):

ABC 123

TAKE **number** will not take anything, but puts the value **0** into the variable, because it finds an alphabetic character before it finds a digit. Consider the following example:

```
extract from input
    take number PosX
    take number PosY
```

The **input** function returns the following:

```
200  -450
-----
^
```

When EASEL OS/2 EE executes the statement:

```
take number PosX
```

the number 200 is taken and copied to the PosX variable. EASEL OS/2 EE skips over the leading blanks. The pointer moves as shown below.

```
200  -450
-----
^
```

Then, EASEL OS/2 EE executes:

```
take number PosY
```

The number -450 is taken as the value of the PosY variable. EASEL OS/2 EE skips over the leading blank. The pointer moves as shown below.

```
200  -450
-----
^
```

Note that these clauses do not take a decimal point or any digits beyond a decimal point. If you want floating point values to be returned, use **float** (do *not* use **number**). However, you can extract an integer number and copy it to a floating point variable; EASEL OS/2 EE will convert the integer to a floating point value.

4.5.3.8 take to or skip to number

Either of these statements starts with the current pointer position and skips or takes blanks up to the first nonblank character that might start an integer number. (Like **take [by] number** and **skip by number**, these clauses do not pass over alphabetic characters.) The pointer is then positioned at the beginning of the number.

If EASEL OS/2 EE cannot find a number, the pointer does not move, and, in TAKE clauses, the value 0 is copied to the specified variable.

4.5.3.9 take [by] or skip [by] float

You use **float** to extract floating point values (a series of digits which may include a decimal point). Either of these statements starts with the current pointer position and skips over any leading blanks and alphabetic characters. It then skips or takes subsequent digits (which may include a decimal point or a leading sign), up to the first nondigit (e.g., a blank, an alphabetic character, or a second decimal point) or the end of the source. The pointer is then positioned immediately after the end of the floating point number.

Consider the following example:

```
extract from input
  take float PosX      # PosX and PosY are floating
  take float PosY      # point variables.
```

The **input** function returns the following:

```
123 45.6 -45
```

```
-----
^
```

When EASEL OS/2 EE executes the statement:

```
take float PosX
```

EASEL OS/2 EE will take the number 123, convert it to 123.00, and copy it to the variable PosX. The pointer moves as shown below.

```
123 45.6 -45
```

```
-----
^
```

Then, EASEL OS/2 EE executes:

```
take float PosY
```

EASEL OS/2 EE skips over the leading blank, and 45.60 is taken and copied to the variable PosY. The pointer moves as shown below.

 ^

If EASEL OS/2 EE cannot find a number, the pointer does not move, and, in TAKE clauses, the value 0.00 is copied to the specified variable.

You can extract a floating point number and copy it to an integer variable; EASEL OS/2 EE will truncate the floating point and convert it to an integer value.

4.5.3.10 take to or skip to float

Either of these statements starts with the current pointer position and skips or takes subsequent characters up to the first nonblank, nonalphabetic character that might start a floating point number (e.g., a digit, leading sign, or decimal point). These clauses *do* pass over alphabetic characters. The pointer is then positioned at the beginning of the floating point number.

4.5.3.11 take [by] or skip [by] TEXT

TEXT is a specified string value. The use of text strings is particularly helpful when you know that certain strings appear in the source, but you do not know in what columns they appear.

Either of these statements starts with the current pointer position, searches for the beginning of the specified string (skipping over any characters that do not match the string), and skips or takes the specified string. The pointer is then positioned at the first character after the end of the string. If the string is not found, the pointer does not move, and, in TAKE clauses, the null string is copied to the specified variable.

Consider the following example:

```
extract from Log
  skip ": "
  take 20 Error
```

Log contains the following:

DIAGNOSTIC: FIELD TOO LARGE

 ^

When EASEL OS/2 EE executes the following:

```
skip ": "
```

EASEL OS/2 EE skips past the specified delimiting string, ": ". The pointer moves as shown below.

DIAGNOSTIC: FIELD TOO LARGE

 ^
Next EASEL OS/2 EE executes:

take 20 Error # uses COLUMN option

EASEL OS/2 EE takes the next 20 characters ("FIELD TOO LARGE ")
as the value of the Error variable.

4.5.3.12 take to or skip to TEXT

Either of these statements starts with the current pointer position, searches for the beginning of the specified string, and then skips or takes characters up to, but not including, the specified string. The pointer is then positioned at the first character in the string.

If EASEL OS/2 EE cannot find the specified string, the pointer does not move. For TAKE clauses, the null string (" ") is copied to the specified variable.

4.5.3.13 take [by] or skip [by] COLUMN (Left-to-Right Movement)

Either of these statements starts with the current pointer position, and skips or takes subsequent characters from left to right over the specified number of columns. The pointer is moved accordingly.

These statements are particularly helpful when your input is column-oriented; that is, when you know that data reside in particular columns in a fixed-field layout. For example:

take 5 VarName

tells EASEL OS/2 EE to take five columns, starting at the current pointer position, and to leave the pointer after the fifth column.

4.5.3.14 take [by] or skip [by] COLUMN (Right-to-Left Movement)

Unlike other kinds of TAKE and SKIP clauses, the COLUMN specification lets you specify a negative number. This moves the pointer to the left of its current position, to the specified column.

When you specify a negative number, either of these statements starts at one position to the left of the current pointer position and moves the pointer by the specified number of columns, from right to left. This takes or skips the characters starting with the ending position of the pointer and extending up to, but not including, the beginning position of the pointer. The pointer is then positioned at the leftmost column to which it was moved. For example:

```
take -5 VarName
```

tells EASEL OS/2 EE to take five columns, starting one character to the left of the current pointer position, and leaving the pointer at the fifth column to the left. If the current pointer position is at column 10, the pointer will be left at column 5; columns 5 through 9 will be taken. Or, for example:

```
take -1 VarName
```

tells EASEL OS/2 EE to take the single column to the left of the current pointer position.

If the specified number of columns does not exist within the source, the pointer does not move, and, in TAKE clauses, the null string is copied to the specified variable.

In the example below, negative column numbers are used in conjunction with a **skip to last** statement. This is helpful when you know that the last data fields in the source contain data that you want to extract.

```
extract from input
  skip to last
  take -5 ID
  take -3 Category
```

In this example, the **input** function returns the following:

```
      X2200059
-----
^
```

When EASEL OS/2 EE executes:

```
skip to last
```

the pointer moves from the first column to the position just after the last column:

```
      X2200059
-----
                ^
```

Next, EASEL OS/2 EE executes:

```
take -5 ID
```

EASEL OS/2 EE takes the last five columns in the source (00059) as the value of the variable ID. The pointer moves as shown below.

```
      X2200059
-----
                ^
```

Next, EASEL OS/2 EE executes:

```
take -3 Category
```

EASEL OS/2 EE takes the previous three columns in the source (X22) as the value of the variable Category. The pointer moves as shown below.

```
      X2200059
-----
      ^
```

4.5.3.15 take to or skip to COLUMN (Left-to-Right Movement)

Either of these statements starts with the current pointer position and skips or takes subsequent characters up to, but not including, the specified column. The designated column number must be in the range 1 to the final character of the source. The pointer is then positioned at the specified column number.

Consider the following example, which uses **skip to**, **skip [by]**, and **take [by]** statements:

```
extract from input
  skip to 7
  take 2 Initials
  take 10 LastName
  skip 1 Sex
  take 3 LaborCategory
```

The **input** function returns the following:

```
002982GRRIMBAUD  F025
-----
      ^
```

When EASEL OS/2 EE executes the following:

```
skip to 7
```

the pointer moves to column 7, as shown below:

```
002982GRRIMBAUD  F025
-----
      ^
```

Next, EASEL OS/2 EE executes:

```
take 2 Initials
```

The letters, GR, are copied to the variable Initials. The pointer moves as shown below:

002982GRRIMBAUD F025

^

Next, EASEL OS/2 EE executes:

take 10 LastName

EASEL OS/2 EE takes the next 10 columns ("RIMBAUD ") as the value of the variable LastName. The pointer moves as shown below.

002982GRRIMBAUD F025

^

Next, EASEL OS/2 EE executes:

skip 1

EASEL OS/2 EE skips the "F" in the input string. The pointer is positioned as shown below.

002982GRRIMBAUD F025

^

Finally, EASEL OS/2 EE executes:

take 3 LaborCategory

EASEL OS/2 EE takes the digits "025" as the value of the variable LaborCategory. The pointer is positioned as follows:

002982GRRIMBAUD F025

^

If the specified column number does not exist within the source, the pointer does not move. For TAKE, the null string ("") is copied to the specified variable.

4.5.3.16 take to or skip to COLUMN (Right-to-Left Movement)

You cannot specify a negative number in either of these clauses; for example:

take to -5 # WRONG

skip to -5 # WRONG

Nevertheless, you can specify a column that is to the left of the current pointer position. When you do so, the pointer is moved from one character to the left of the current pointer position to the specified column, and then

skips or takes the characters from the specified position up to, but not including, the starting pointer position. For a TAKE clause, the values are copied from left to right to the specified variable. For example, if the cursor is at column 5 and you specify:

```
take to 2 VarName
```

EASEL OS/2 EE takes the characters in columns 2, 3, and 4.

4.6 Arrays: Defining and Referencing Groups of Constants or Variables

An array is a group of variables or constants, all accessed by the same name but with different subscripts. An array element is used like a regular string, integer, floating point, or boolean variable or constant, except that you cannot append anything to an array element. The size of an array cannot exceed 64K, therefore, the maximum number of elements per array depends on the type of array, as follows:

16K	integers or strings
8K	floats
512K	booleans

Model

```
[global] { integer | string | float | boolean }
[ variable | constant ]
{ ARR_NAME[ N ] |
  ARR_NAME[ N1[, N2]... ] |
  ARR_NAME[ NA thru NB ]
  ARR_NAME[ N1A thru N1B[, N2A thru N2B]... ] } [is
  VAR1[, VAR2]... ]
```

When defining an array, you must specify the data type, the array name, and the range of permissible subscripts (dimensions) of the array. The subscripts for arrays are always surrounded by brackets []. In the example:

integer SalesGraph[2,3]

integer is the data type, **SalesGraph** is the array name, and [2,3] defines the dimensions of the array. Thus:

- the array is two-dimensional
- the first dimension has two elements
- the second dimension has three elements
- the array contains a total of six elements (2*3)

4.6.1 Specifying Dimensions in an Array Definition

An array can have up to 11 dimensions (although you will probably not use more than three). Only the first dimension can be resized, using the **resize** action statement.

For each dimension specification, you can specify what integer values you want to use to reference the elements. For example,

`string Colors[0 thru 5]`

has one dimension containing six elements, starting with “0” and ending with “5.” If you do not specify a starting reference, the default is 1. For example,

`integer SalesGraph[4]`

has four elements, starting with “1” and ending with “4.”

Values for specifying dimensions can be integer constants, integer literals, or expressions containing both. Variables cannot be used. For example, if you have defined a constant such as:

`constant NumDataPoints is 10`

and then later define an array as:

`integer Graph_A[NumDataPoints + 5]`

the array `Graph_A` will have 15 elements.

4.6.2 Specifying Elements in an Array Definition

Initial values may be given to array elements. For example, if an array is defined as:

`integer Answers[4] is 5,17,8,2`

the initial values of the elements are 5, 17, 8, and 2. If no values or too few values are given, the uninitialized elements are given the default values:

Element	Defaults To
boolean	false
integer	0
float	0.00
string	"" (null)

If too many values are given, the extra values are dropped. In either case, a warning message is generated when you compile the program.

Following are examples of array definitions:

```
integer Bar[0 thru 2,10 thru 12] is  
    2,3,3,Var_A,15,(5*Var_A),5,3,1228
```

```
string Sales[2,3] is "Boston","New York",  
    "Chicago","London","Phoenix","Paris"
```

```
boolean Bar[0 thru 2,6 thru 7] is  
    true,true,true,false,false
```

```
float Sales[1,6] is 15.5,14.8,16.7,20.3,18.9,21.7
```

When EASEL OS/2 EE initializes an array, it cycles through all the subscripts in the array, changing the last subscript first. Using the example:

```
integer SalesGraph[2,3] is 1,2,3,4,5,6
```

the subscripts are cycled through and assigned initial values in the following order:

```
SalesGraph[1,1] is assigned the value 1  
SalesGraph[1,2] is assigned the value 2  
SalesGraph[1,3] is assigned the value 3  
SalesGraph[2,1] is assigned the value 4  
SalesGraph[2,2] is assigned the value 5  
SalesGraph[2,3] is assigned the value 6
```

4.6.3 Global Variable Arrays

You can define a global variable array by beginning the definition with the keyword **global**. For example:

```
global integer Profits[5]
```

When global arrays are loaded from global space into program space for an EASEL OS/2 EE program that is executed by a **change to program** statement, the new number of dimensions is not checked against the number of dimensions defined for the array in the original EASEL OS/2 EE program. Therefore, if you dimension the array in the new program to be larger than was defined in the original program, then the last elements will contain default values for the constant or variable types. If you dimension the array in the new program to be smaller than was defined in the original program, the last elements will be dropped.

4.6.4 Referencing Elements in Arrays

When referencing an array, the [N] becomes the subscript that references the array element, and does not designate array dimensions as it did when you defined it.

When referencing a multiple dimension array, the [N1[, N2]...] becomes the subscripts for each dimension of the array that references a single array element.

For example, if you defined an array as:

```
integer Apples[2,2] is 1,15,30,45
```

and then later specified in your program:

```
copy Apples[1,2] to Accumulate
```

EASEL OS/2 EE will copy **15** to Accumulate.

4.6.5 Resizing an Array

Arrays are resizable in the first dimension only. The **resize** statement lets you initially define an array with one range of subscripts, and then later make it larger or smaller. If the **resize** enlarges the array, existing data is not disturbed, but if it decreases the array size, some array space will be dropped and the corresponding data will be lost.

When you increase the size of an array, additional memory is used. The **resize** statement in the *IBM EASEL Version 1.1 for OS/2 Extended Edition Language Reference Manual* shows some common sizes of arrays, and 11.1.6, "Controlling Flow Within a Program" on page 11-13 shows you how to create additional space if there is not enough memory.

The following example assumes that most users will need to manipulate only ten objects, each object with three values:

```
integer variable ObjectLimit is 10
```

```
string ObjectChildren[1 thru ObjectLimit,3]
```

If, however, the user needs to manipulate more than ten objects, the **resize** action statement can be used to resize the array.

Model

```
resize array ARR_NAME to DIMENSION
```

For example:

```
action RedimensionArray is
  copy (5+ObjectLimit) to ObjectLimit
  resize array ObjectChildren to ObjectLimit
```

Here, ObjectChildren is increased by 5 when more space is needed. Note that only the first dimension (the number of objects, initially "1 thru

ObjectLimit") is increased (from 10 to 15, the first time the action is invoked), and that each object still can have only three values.

4.7 Built-in Functions

EASEL OS/2 EE's predefined built-in inquiry functions allow you to inquire about the attributes of an object or the nature of a stimulus. For example, you might use a function to find the current position of an object's graphics cursor, or to determine the specific text of an input that triggered a response from the keyboard. As the answer to your inquiry, EASEL OS/2 EE returns a value for the function, based upon the position or contents of the object, or the nature of the stimulus.

All built-in function names are keywords and therefore begin with lowercase letters. Since built-in functions return values, they can be used anywhere a value of the appropriate data type is expected, such as in expressions and object definitions.

The following sections give an overview of all the built-in functions. All the functions are described in detail as they relate to particular features in each chapter. Already discussed in this chapter were the **marker** function as related to the **extract from** statement, and the **length of** function as related to string variables.

There are five categories of built-in functions:

- Response inquiry functions allow you to inquire about the characteristics of a stimulus that produced a response.
- Object inquiry functions allow you to inquire about the characteristics of a particular object.
- Item inquiry functions allow you to inquire about the characteristics of action bar items.
- Action inquiry functions allow you to determine whether certain textual region statements were executed successfully.
- Special inquiry functions allow you to determine the current time or date, the amount of free memory, the members in a class of objects, the position of the pointer in an **extract from** statement, or the length of a string variable.

4.7.1 Response Inquiry Functions

The response inquiry functions are:

ancestry	eventparam	output	xcoord
eventnumber	input	parameter	ycoord

These functions are used to discover the characteristics of the stimulus that initiated a response. For example, if the response was stimulated by selecting an object, the value returned by the **object** function is the name of

the object that was selected, and the values returned by the **xcoord** and **ycoord** functions are the positions of the selection point in the object's coordinate system. The ancestry of the selected object is returned by the **ancestry** function, and the parameter attribute value of the selected object is returned by the **parameter** function. If the response was stimulated by input from the keyboard or an application, the value returned by the **input** function is the data that was received by EASEL OS/2 EE from the keyboard or the application. The **eventnumber** and **eventparam** functions are used in conjunction with the **response to stimulus** statement.

4.7.2 Object Inquiry Functions

Object inquiry functions are used to discover a specified object's attributes and position. These functions are:

ancestry of	left [of]	window xposition [of]
background at	line size of	window xsize [of]
background of	parameter of	window yposition [of]
border of	priority of	window ysize [of]
bottom [of]	right [of]	xcursor [of]
checked in group	selectability of	xmiddle [of]
column size of	selected line from	xposition [of]
exists	text of	xsize [of]
foreground at	textual	ycursor [of]
foreground of	textual line from	ymiddle [of]
handle of	top [of]	yposition [of]
is checked	visibility of	ysize [of]

The functions that return integers can be used as operands in arithmetic expressions, or can be compared to integer values in relational expressions. The functions that return strings can be used anywhere a string value is expected.

Be sure to define an object's position (and ancestry, if there is any) before you apply any object inquiry function to that object. If you have a response that deletes a parent object, you cannot reference any of its children, because when a parent is deleted, so are its children. Therefore, if you try to reference any of the inquiry functions for those deleted children, you will receive an error message.

The object inquiry functions **xsize**, **ysize**, **line size of**, and **column size of** require internal calculations that are not made until runtime, and therefore cannot be included in object definitions. For example, the following statements would return incorrect results at compile time, because **xsize** and **ysize**, which calculate the coordinates of the region News, are in an object definition:

```
textual region News window size 10 columns 10 lines
    at position 100 200
```

```
green graphical region RegionA
    size (xsize of News) (ysize of News) ... # WRONG
```

To include a function in the definition of an object, create the object in an **add** action statement (which is executed at runtime), such as:

```
textual region News window size 10 columns 10 lines
    at position 100 200
```

```
response to start
    add region
        RegionA size (xsize of News) (ysize of News) ...
```

For those object inquiry functions listed above that are followed by the optional keyword [**of**], you can omit the object name when the function is specified in a drawing statement. In this case, the object defaults to the current object. For example:

```
add to Blue
    move to top # equivalent to "top of Blue"
    :
```

or:

```
add region Blue ...
    move to xmiddle # equivalent to "xmiddle of Blue"
    :
```

For clarity, always specify the object name for the function, even if the syntax allows you to omit it. For example:

```
add to Blue
    move to top of Blue
    :
```

4.7.3 Item Inquiry Functions

Item inquiry functions are used to discover a specified action bar item's attributes. These functions are **is checked**, **is enabled**, and **text of item**.

4.7.4 Action Inquiry Functions

You can test whether an **insert file** drawing statement, or a **read** or **write** action statement, was successful by using the **ioerror** action inquiry function. You can test whether a **find** action statement was successful by using the **found** action inquiry function.

You can test whether an error code was returned by an external subroutine or an error occurred with the **start** statement or **send** statement by using the **errorlevel** action inquiry function. An **errorlevel** value of 0 indicates that no error code was generated.

4.7.5 Special Inquiry Functions

A number of special inquiry functions are provided. They are

clipboard	length of	members of
date	marker	selected line
freesize	member	time

4.7.5.1 The date and time Functions

You can use the **date** or **time** function to find the current date or time, and display it or use it as a date or time stamp. The value returned for the **date** function is the current date as MM/DD/YY, where:

MM is the month, and appears as 1 or 2 digits
DD is the day, and appears as 1 or 2 digits
YY is the year, and appears as 2 digits

For example, if you specify:

```
response to CheckDate
  change Calendar to
    date
```

EASEL OS/2 EE returns the current date, such as:

9/4/89

The value returned for the **time** function is the current 24-hour time as HH:MM:SS, where:

HH is the hour, and appears as 1 or 2 digits
MM is the minute, and appears as 2 digits
SS is the second, and appears as 2 digits

For example, if you specify:

```
response to CheckTime
  change Time to
    time
```

EASEL OS/2 EE returns the current time, such as:

17:07:00

To get the date or time in the format specified by the user in the PM Control Panel or CONFIG.SYS COUNTRY setting, use the **LocalDate()** or **LocalTime()** function from the Date Library.

4.8 Expressions, Operators, and Operands

An expression is a combination of operands and operators that produces an integer, floating point, string, or boolean value as a result. It can be specified wherever a value is required.

Valid operators are arithmetic, relational, or boolean. You can specify operators in all of these categories in a single expression.

Valid operands are literals, constants, variables, and built-in functions. You must specify operands that are either of the type required for the particular operator, or that EASEL OS/2 EE can convert to the correct type (see 4.9, "Type Conversions" on page 4-53.) For example, if you specify multiplication (the operator *), the operands must be integers or floating point numbers, such as:

```
(A * 56)
```

where A is an integer constant or variable. If you specify an operand that EASEL OS/2 EE cannot convert, an error message is produced at compile time. For example:

```
(true = 1) # Integers cannot be converted to booleans.
```

All expressions must be enclosed in parentheses. You can specify expressions wherever a compile-time value is required; if the expression contains one or more variables, the values of the variables are evaluated at compile time.

The simplest expression is a literal, such as (-10). The name of a variable or constant can also be considered an expression, when it is enclosed in parentheses. More complex expressions are formed by combining operators and operands; for example:

```
(Width * 5)
```

or:

```
(A > B)
```

In still more complex expressions, called *compound* expressions, the operands are themselves expressions; for example:

```
(Width = ((Height / Xfactor) + 8) * 5)
```

(Note that the value of this expression is **true** or **false**, and that the contents of the variable Width are not changed.)

The following is an example of a fairly complex expression. It could be used to place one of a series of objects in some incremental position on the screen.

```
integer constant   KeyOffset is ...
                  KeyLevel is ...
                  KeyWidth is ...
                  KeyHeight is ...
                  Separation is ...
:
key Done at position (KeyOffset*3+Separation) KeyLevel
  box KeyWidth KeyHeight
  move (KeyWidth/2) ((KeyHeight/2) + 5)
```

In this example, the expression:

```
(KeyOffset * 3 + Separation)
```

means multiply 3 times the value of the constant KeyOffset, and add the product to the value of the constant Separation.

4.8.1 Arithmetic Operators

You can specify the following arithmetic operators in an expression:

Operator	Meaning	Example
Unary:		
—	Minus	-5
Binary:		
*	Multiplication	Base * 100
/	Division	Total / Unit
mod	Modulo (remainder of integer division)	C mod 12
+	Addition	A + 10
—	Subtraction	Num2 - 8

The unary minus operator simply reverses the sign of the value specified as the operand, and generates an integer or floating point result. Each of the binary operators arithmetically combines two operands and generates an integer or floating point result.

You can specify any value as an arithmetic operand, provided its type is integer or floating point, including the following:

- An integer or floating point *literal*
- A *variable* or *constant* that contains an integer or floating point value
- An *expression* that has an integer or floating point result

- A *built-in function* that returns an integer value

Operands in arithmetic expressions must be integer or floating point values. Do not specify string or boolean values with arithmetic operators; if you do, EASEL OS/2 EE generates an error message and does not convert the value.

When the division operator (/) is used with an integer operand, but assigned to a floating point variable or constant, a floating point value results; for example:

copy (3/2) to FP_Var # 1.00 will be copied to FP_Var.

The actual computation in this expression is performed with integer math, because all operands are integers. If one or more of the operands is a floating point value, a floating point computation is performed. For example:

copy (3/2.0) to FP_Var # 1.50 will be copied to FP_Var.

copy (3.0/2) to FP_Var # 1.50 will be copied to FP_Var.

The **mod** operator can only be used with integer values; otherwise, an error message will be produced.

4.8.1.1 Precedence in Arithmetic Operators

When EASEL OS/2 EE evaluates an expression that contains arithmetic operators, it does not necessarily proceed from left to right in performing the necessary arithmetic. Instead, it proceeds according to an established precedence of operations. Precedence is the order in which operators are evaluated. The following lists the order of precedence, from highest to lowest:

1. **-** (unary minus)
2. ***** / **mod**
3. **+** **-** (binary; addition and subtraction)

When precedence cannot be used to determine the order of evaluation—that is, when operators have equal precedence—EASEL OS/2 EE proceeds from left to right. In the expression:

(A / 5 * B)

EASEL OS/2 EE divides A by 5 and then multiplies the quotient by B.

Consider the following example:

(-5 + 2 * 3)

Because the unary minus operator (-) has the highest precedence, EASEL OS/2 EE performs the - operation first; negating 5 produces the result, -5. EASEL OS/2 EE next performs the multiplication operation (*):

$(2 * 3 = 6)$

EASEL OS/2 EE then adds the value, -5, to the value, 6; the result of the expression is 1.

You can use parentheses to clarify expressions and to override the precedence of operators. Consider the expression:

$(A + 7 / D)$

In evaluating this expression, EASEL OS/2 EE first divides 7 by D and then adds the result to A. If, instead, you want the sum of $A + 7$ to be divided by D, you must specify:

$((A + 7) / D)$

If an expression contains nested parentheses (parentheses within parentheses), EASEL OS/2 EE always begins with the operation within the *innermost* set of parentheses.

4.8.2 Relational Operators

Relational operators allow you to compare two values to determine the relationship between them. You can specify the following relational operators in an expression:

Operator	Meaning	Example
=	Equal to	$(A = \text{"Boston"})$
!=	Not equal to	$(\text{Slope} != 10)$
<	Less than	$((B * 50) < C)$
<=	Less than or equal to	$(\text{Debts} <= \text{Assets})$
>	Greater than	$(5000 > \text{Curve})$
>=	Greater than or equal to	$(X_{\text{tan}} >= X_{\text{sin}})$

Each of these operators compares two operands and generates a boolean value (**true** or **false**) as a result. For example, the expression:

$(5000 > \text{Curve})$

is **true** if 5000 is greater than the value of the integer variable, Curve, and **false** otherwise. The expression:

$(A = \text{"Boston"})$

is **true** if the value of variable A is the string, "Boston". Note that when you compare string values, blanks are taken into account. If the value of A is the string, " Boston", the expression is **false**. Similarly, if the value of A is the string "Boston ", the expression is **false**.

With a relational operator, EASEL OS/2 EE allows you to specify any values as operands, provided that the types of operands match. If one of your oper-

ands has a string value, the other also must have a string value. If one of the operands has an integer value, the other also must have an integer value. You cannot specify an expression such as:

```
("0" = 0)      # WRONG
```

4.8.2.1 Precedence in Relational Operators

All relational operators have equal precedence; they have a lower precedence than arithmetic operators. When you combine arithmetic and relational operators in a single expression, EASEL OS/2 EE performs the arithmetic first. For example, the expression:

```
(Debts + Taxes <= (Salary + Savings) * Inflation)
```

is **true** if the sum of Debts plus Taxes is less than or equal to the product of Inflation times the sum of Salary plus Savings.

4.8.3 Boolean Operators

Boolean operators allow you to perform logical operations. You can specify the following boolean operators in an expression:

Operator	Meaning	Example
not	Result is true if operand is false	(not (A > B))
and	Result is true if both operands are true	((V1 = 5) and (V2 != 10))
or	Result is true if either or both operand(s) are true	((Slope < Arc) or (A <= B))

With a boolean operator you must specify boolean values as operands. Most often, these values are the results of operations involving relational operators, as shown in the examples above.

4.8.3.1 Precedence in Boolean Operators

EASEL OS/2 EE evaluates boolean expressions according to an established precedence of operators. The following lists the order of precedence, from highest to lowest.

1. **not**
2. **and**
3. **or**

Boolean operators have a lower precedence than arithmetic or relational operators. When you combine boolean, arithmetic, and relational operators in a single expression, EASEL OS/2 EE applies the arithmetic operators,

then the relational operators, and finally the boolean operators. For example, consider the expression:

$(A > (B - 5) \text{ and } B < C \text{ or } D * 2 = E)$

Suppose that:

$A = 15 \quad B = 20 \quad C = 25 \quad D = 30 \quad E = 30$

Thus the expression becomes:

$15 > (20 - 5) \text{ and } 20 < 25 \text{ or } 30 * 2 = 30$

EASEL OS/2 EE first performs the arithmetic:

$15 > 15 \text{ and } 20 < 25 \text{ or } 60 = 30$

Next, EASEL OS/2 EE performs the comparisons specified by the relational operators:

false and true or false

EASEL OS/2 EE then performs the boolean operations. Because the **and** operator has higher precedence than the **or** operator, EASEL OS/2 EE first evaluates the **and** expression, getting the result **false**:

false or false

(Because one of the operators is **false**, the result of this entire expression is **false**.)

EASEL OS/2 EE then performs the **or** operation:

false

(Because neither operand is **true**, the result is **false**.)

4.9 Type Conversions

When you specify a value in an EASEL OS/2 EE statement, the value is expected to be of the type required by that statement. However, if you specify a value of a type that is not expected in the statement, EASEL OS/2 EE attempts to convert this value to the type required by the statement. For example, suppose you specify:

move to A B

A and B are string variables; A has a value of "0" and B has a value of "200." EASEL OS/2 EE converts the string values "0" and "200" to the integer values 0 and 200, and then performs the move.

Type conversions are performed only on the following values:

Value	Converts to...
boolean	integer, float, string
string	integer, float, Identifier (name) (with and without ancestry)
float	integer, string
integer	float, string
Identifier	string (names) (with and without ancestry)

Other conversions are not performed by EASEL OS/2 EE. *They are never performed inside an expression.* If you specify a value that cannot be converted to the required type, EASEL OS/2 EE produces an error message.

4.9.1 String/Integer and String/Floating Point Conversions

EASEL OS/2 EE will convert string values to integer or floating point values, and vice versa. Therefore, you can provide a string value in any statement in which an integer or floating point value is required, and vice versa. The following are valid specifications:

string variable Pos is "50"

disabled key Stop at position Pos Pos
draw to Pos 50

When a string contains only digits, EASEL OS/2 EE converts it to integers or floating point values in the obvious way ("50" is converted to 50; "50.01" is converted to 50.01). When the string contains blanks or alphabetic or special characters, EASEL OS/2 EE starts at the beginning of the string and proceeds according to the following rules:

- It ignores any leading blanks.
- It examines the first nonblank character(s) in the string.

For integers, if the characters are digits (optionally preceded by a minus sign), EASEL OS/2 EE interprets the value as being an integer.

For floating point values, if the string contains a decimal point, the digits before the decimal point and after the decimal point up to the first nondigit are regarded as the floating point value.

- It terminates at the first character that is not a digit (or decimal point, for floating point values). If it terminates before locating a number, the value is zero.

The following are some sample conversions:

String to Integer		String to Floating Point	
"52"	52	"52.5"	52.50
"-52"	-52	"-52.5"	-52.50
" 52"	52	" -52.5"	-52.50
"52X5"	52	"52.5.1"	52.50
"5 2"	5	"52 5"	52.00
"X52"	0		

As mentioned earlier, EASEL OS/2 EE will not perform conversions inside expressions; therefore, you cannot specify string values as operands for arithmetic operators. For example:

(Pos / "10")

is invalid.

4.9.2 Boolean/String Conversions

When a boolean value is used where a string value is expected, the value **true** will be converted to the string value "1", and the boolean value **false** will be converted to the string value "0".

4.9.3 Integer/Floating Point Conversions

When a floating point value is used where an integer is expected, the floating point value will be truncated at the decimal point; the value will not be rounded. When an integer value is used where a floating point is expected, an implicit type conversion will take place (e.g., the integer 10 will become the floating point 10.00).

If an expression contains both integer and floating point values, all integer values will be converted to floating point values before the expression is evaluated. To obtain the precision desired when using floating point variables in expressions, it is better to specify:

copy (5.0/2) to FloatNum # 2.50 is copied to FloatNum

instead of:

copy (5/2) to FloatNum # 2.00 is copied to FloatNum

4.9.4 String/Name Conversions

EASEL OS/2 EE can convert an identifier (name) to a string, and vice-versa. For example, you can specify:

```
string variable X
:
invisible key BigShape at position 50 100
:
response to Class
  if (object = BigShape) then
    :
  end if
```

In this example, EASEL OS/2 EE converts the object name, BigShape, to the string, "BigShape," so that a comparison can be performed with the string value returned by the built-in function **object**.

4.9.5 String/Name with Ancestry Conversions

The string-to-name and name-to-string conversions discussed above are performed in a less obvious manner when an identifier (name) specifies ancestry.

EASEL OS/2 EE can convert a name with ancestry, such as "PanicButton in Panel", to a string. For example, you can specify:

```
string variable TempName

region Panel size 50 100 at position 100 200

key PanicButton at position 50 50 in Panel

response to start
  copy PanicButton in Panel to TempName
```

When EASEL OS/2 EE executes the **copy** statement, it converts the object name "PanicButton in Panel" to a string, and stores the value in the variable TempName. The value stored in TempName is not, however, "PanicButton in Panel", but is another string that is a more convenient representation for EASEL OS/2 EE: "Panel/PanicButton". You can then use the variable TempName in statements that require an object name as a value; for example:

```
make TempName visible
```

At this point, the value of TempName is converted back to the name of the object, "PanicButton in Panel", and this object is made visible.

The following example is incorrect:

```
response to ...
  copy "PanicButton in Panel" to TempName
  :
  disable TempName      # WRONG
```

In this example, EASEL OS/2 EE does not disable the object PanicButton in Panel. The string, "PanicButton in Panel", is not equivalent to the name, PanicButton in Panel, and therefore cannot be converted. Instead, you would specify:

```
response to ...
  copy "Panel/PanicButton" to TempName
```

4.10 Modules

EASEL OS/2 EE includes a library of the following modules:

Name	Use
M3270	Performs 3270 communications.
M5250	Performs 5250 communications.
CText	Reads colored ASCII files into colored textual regions.
BGraph	Formats data and creates business graphics.

Each module is a set of action subroutines that perform commonly used actions. Modules can save you considerable programming time; instead of writing the action statements yourself, you can simply reference the appropriate module in your program.

You reference a module using the **module** environment declaration, shown below. This makes the action subroutines that comprise the module available in your EASEL OS/2 EE program. You can reference any number of modules in an EASEL OS/2 EE program.

Model

```
module { BGraph | CText | M3270 | M5250 }
```

When you reference a module, each action statement that is part of the module is available in your program exactly as though you had specified its definition in the program. Arguments are passed to the module by storing data in variables defined by the module. Similarly, outputs from the module are stored in variables that it defines, and these variables can be referenced in your program.

Modules are stored in the module subdirectory of your main EASEL OS/2 EE directory. Each module has both a public file and a private file. The public file (which has the extension .PUB) is a quick reference on how to use the module, and defines the variables that you use to pass information to and from the module's subroutines.

CAUTION:

You should examine the variable names defined in the public part of the module to ensure that you don't accidentally use the same names elsewhere in your program.

The private file (which has the extension .PRI) contains the code that performs the actions.

CAUTION:

Do not modify this directory or any files in it. Doing so may make your programs incompatible with future releases.

Note: You cannot create and store your own modules in the EASEL OS/2 EE module library. Instead, you can write a dynamic link library or local application (see 25.2, “Writing Library Functions and Subroutines” on page 25-3 and 25.1, “Writing Well-behaved Local Applications” on page 25-2).

Chapter 5. Objects, Coordinate Systems, and Fonts

5.1 Types of Objects and Their Uses

With EASEL OS/2 EE, you can create, display, and manipulate objects. There are eight categories of objects:

- A **key**, which contains graphical and/or textual contents. When enabled, the user can select it to stimulate a response; when disabled, it provides information.
- A **graphical region**, which is like a key except that it has a rectangular area in which to display its contents, and it can have PM window attributes, such as a title bar, scroll bars, and so on.
- An **image region**, which contains bitmap images.
- A **sense region**, which is like a graphical region except that it has no graphical contents or visual appearance.
- A **textual region**, which is like a graphical region except that it can only contain text, and cannot contain any other objects.
- A **dialog region**, which is like a graphical region except that it can only contain dialog control objects.
- A **dialog box**, which requests information from the user. Dialog boxes are made up of dialog control objects.
- A **Dialog Control**, which is inside a dialog box or dialog region.

5.2 The Components of an Object

Each object must be defined in an *object definition*. The syntax model for a key is shown below. (For details on defining a key or graphical region, see Chapter 7, “Graphical Objects, Image Regions, and Sense Regions”; for details on defining a textual region, see Chapter 8, “Textual Regions”; for details on defining dialog boxes, dialog regions, and dialog controls, see Chapter 9, “Dialog Boxes, Dialog Regions, and Dialog Controls” on page 9-1.)

Model (for a key graphical object)

```
[ enabled | disabled ] [ visible | invisible ] [ [color] COLOR]
key KEY_NAME
  at { [position] X Y | default position }
  [in PARENT_NAME1 [in PARENT_NAME2]... ]
  [priority is PRIORITY_NO]
  [parameter is PAR_STRING]

  [DRAWING_STATEMENT]...
```

Each object definition consists of two parts: an *object statement*, which specifies the object's type, name (identifier), attributes, and position; and a set of *drawing statements* which describe the object's initial contents, or appearance. For example:

```

      attributes
      /
enabled visible red key Hi at position 10 10
      /      /      /
      type  name  position
      /
box 100 100 # drawing statement
move 50 50 # drawing statement
text center "GO" # drawing statement
```

The object definition contains initial attributes of the object. Attributes are the characteristics of the object—for example, selectability, visibility, and color. Region attributes describe the windowing appearance of the object.

The *type* of object is specified as a keyword (**key**, **[graphical] region**, **[colored] textual region**, and so on). The object *name* is a unique identifier you give the object, for referencing in the program. The *position* is the object's location within its parent on the screen; the default parent in a win-

dowed application is the primary region; in a fullscreen application it is **master**.

Drawing statements describe the appearance of the object. You can specify as many drawing statements as you need to generate the desired object. Drawing statements perform such operations as drawing a line or a circle, or specifying text for display.

Note: You cannot specify all drawing statements for all object types; for example, you cannot specify “box” for a textual region.

You *must* specify the object type, name, and position. For regions, dialog boxes and controls, you must also define the size. You can leave all other attributes and contents initially unspecified.

The attributes and contents that you provide when you define an object represent the initial attributes and appearance of the object. Both attributes and contents can be changed during runtime, using action statements that are part of a response definition.

5.3 Common Object Attributes

The following sections discuss attributes that most objects have in common. See 10.1, “Region Attributes” on page 10-2 for a discussion of attributes that are common to regions.

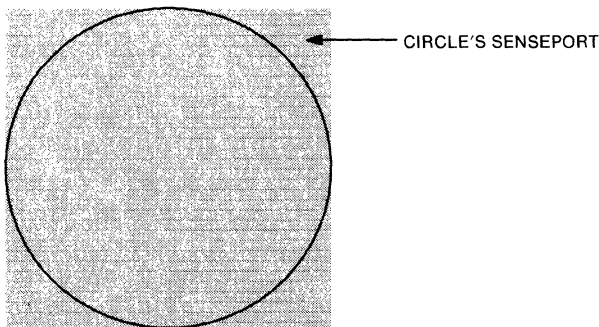
5.3.1 Selectability

The selectability attribute determines whether the user can stimulate a response by selecting the object on the screen. An object is either selectable or not. You can specify **enabled** (selectable) or **disabled** (not selectable) as an attribute when you define the object. If you define an object as being **enabled**, and you have defined a response for that object in the program, selecting the object causes the actions specified in the response to be performed. If you define an object as **disabled**, selecting the object does not produce a response, even if you include a response for that object in the program.

5.3.1.1 Object Senseport

Sometimes a selectable object contains several distinct lines or shapes, perhaps with empty space between. When users select the object to invoke a response, they might select one of these lines or shapes or the space between them.

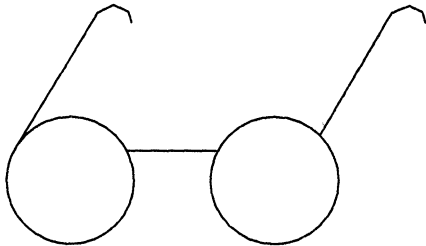
An object can be selected anywhere in its *senseport*. The senseport is the smallest rectangular portion of the screen that completely encloses the displayed portion of a selectable object, and all of its children. For example, the selectable object below is the circle:



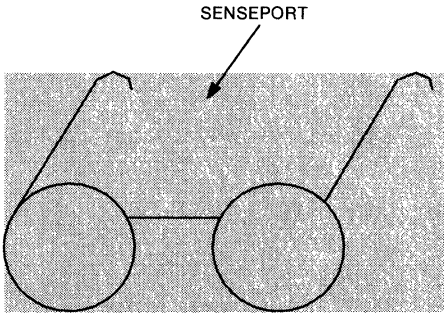
The shaded area is its senseport. Selecting anywhere within the shaded area (either inside or outside the boundaries of the circle) triggers a

response to the object, Circle. The senseport is not actually displayed; it is simply the smallest rectangular area encompassing each object and its children.

Suppose the following is drawn:

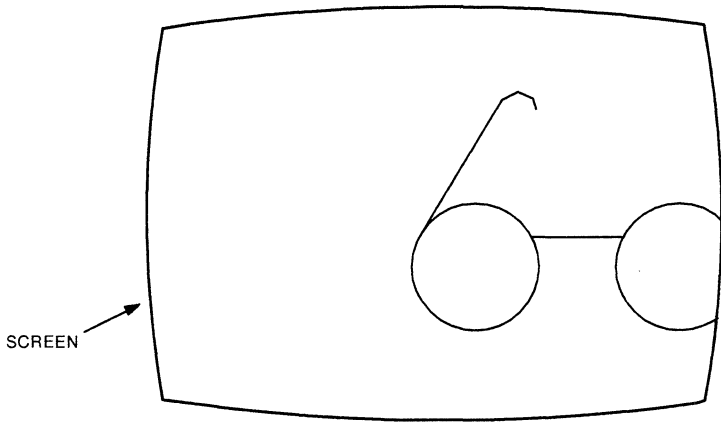


The senseport for this key is the smallest rectangle that completely encloses the entire image:

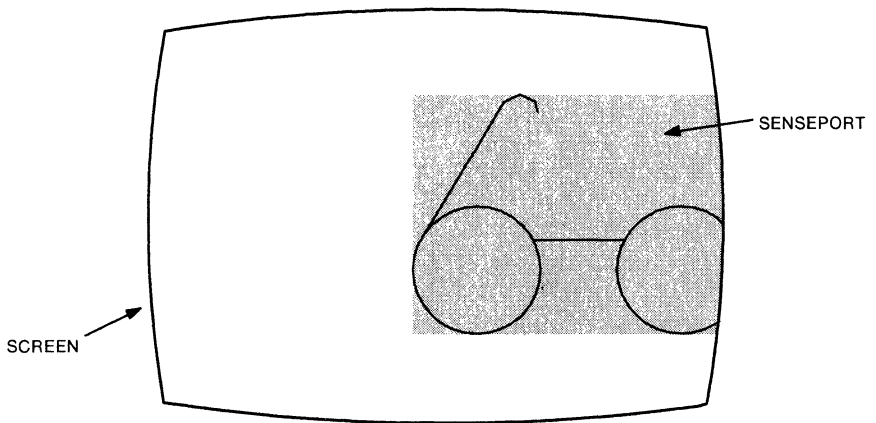


If the object is enabled and a response has been defined for this object, then selecting anywhere within the shaded area, even in an area not occupied by any graphics, triggers a response.

Perhaps only a portion of a selectable object is visible on the display screen. Suppose the visible display is the following:



In this case, the senseport is the shaded area shown below:



You can change the selectability of objects during runtime using the **enable** and **disable** action statements. For example, you might enable keys only when it is meaningful for the user to select these keys.

5.3.2 Visibility

The attribute of visibility determines whether the object will be displayed on the screen. If you define an object as **visible**, it is displayed (if allowed by its position and priority). If you define an object as **invisible**, it is not displayed.

There are some circumstances under which you will not be able to see an object on the screen, even if it is defined as visible:

- A visible object is not displayed if its position and contents are outside the boundaries of the screen.
- A visible object is not displayed if its parent is invisible. Whenever an object is invisible, or becomes invisible in a response definition, none of its children are displayed even if they have the attribute of visibility. Defining an object as visible does not automatically make its children visible. Only the **visible** children are displayed.
- If a visible object is the child of a region, the object is not displayed if its position and contents are outside the boundaries of the window size specified for the region.
- A visible object might lie partially within the screen, or partially within its graphical region's window. In that case, the part of the object that lies within the screen is drawn and the other part is not drawn.
- The primary region is made invisible.

You can change the visibility of objects during runtime using the **make visible** and **make invisible** action statements, to control the user's options and eliminate the possibility of user mistakes. For example, an object that is currently distracting or useless can be made invisible until a stimulus occurs that makes the object useful again. When that happens, the object can be made visible as part of the response to that stimulus.

5.3.3 Color

The attribute of color specifies the color in which EASEL OS/2 EE draws the object. Every object has a foreground color; this is the color in which the contents of keys are drawn. If no foreground color is specified, the foreground color is white, except for dialog boxes and dialog controls, which use the colors defined by Presentation Manager (PM). In the statement:

```
green key Go at position 10 10
```

green is specified as the (foreground) color of the key.

(When you define a region, you can specify a color for both the foreground and the background. This is discussed below in more detail.)

Specify a color by using a color keyword, such as **red**, or the **color** keyword followed by a string value representing the color (such as **red**) or by an integer value representing the color number (such as "1").

5.3.3.1 Standard EASEL OS/2 EE Color Set

The standard EASEL OS/2 EE color keywords and their corresponding numbers are:

Color Keyboard	Color Number
red	1
green	2
yellow	3
blue	4
magenta	5
purple	5
cyan	6
aqua	6
white	7
black	8

5.3.3.2 Additional EASEL OS/2 EE Colors

These colors can be changed for fullscreen programs only.

Color Number	Windowed Color	Fullscreen Default Color (EGA/VGA)
18	dark gray	light gray
19	dark blue	dark gray
20	dark red	light blue
21	dark purple	dark purple
22	dark green	bright pink
23	turquoise	orange
24	mustard	gold
25	light gray	brown
26	window background	window background
27	window text	window text

Colors 26 and 27 are set in the PM Control Panel. These colors cannot be used in a **colored textual region**.

Whether you specify a color number, string value, or integer value for the color value, you must specify the keyword **color**. For example, you can specify:

```
color "red" key RedKey at position 40 50
```

```
color 1 key A at position 20 20      # 1 is the color
                                     # number
                                     # corresponding to
                                     # the color red.
```

```
color UserChoice key A ...          # UserChoice may be a
                                     # variable or
                                     # constant
                                     # representing a
                                     # keyword or number.
```

You can also use the keywords **red**, **blue**, and so on. For example:

```
red key A at position 50 50         # also OK
```

But you cannot specify:

```
1 key A ...                         # WRONG: missing keyword color
```

```
UserChoice key A ...                # WRONG: missing keyword color
```

```
"red" key RedKey ...                # WRONG: missing keyword color
```

However, you can specify:

```
red key RedKey                      # no need for color keyword
```

5.3.3.3 Color in Regions

Each region has a **foreground** color, a **background** color, and, if you specify a color border, a **border** color. You can specify each of these colors in the region's definition.

The background color is the color of the field on which the contents of a region are displayed. For example, the following definition defines a graphical region with a yellow background:

```
yellow graphical region Help size 10 25 at position 0 0
```

No border is specified, so the region is displayed as a solid yellow rectangle. The contents of the region are drawn on top of this yellow background. (If you do not specify a background color, the background will be drawn in the default color of black.)

The foreground color is the default color for the region's drawing statements. Any drawing statement in the contents of the region that does not have its own color specification is drawn in the foreground color. For example, you can specify a foreground color by defining a graphical region such as:

```
graphical region Small size 20 50 at position 200 200
    aqua foreground
    move to 10 25
    circle radius 5
```

This **circle** drawing statement, which does not have a color specification, is drawn in aqua, the color specified for the graphical region's foreground. (If you do not specify a foreground color, the circle will be drawn in white, the default color for the foreground of the region.)

For textual regions, the foreground color is the color of all the text. If you want to specify different colors for individual characters, rather than just one color for the entire text, you must define and use a **colored textual region**.

You can change any or all of the colors in an object during runtime (see 6.3.4.3, "Changing Color" on page 6-29). You can change the foreground of a key and you can change the foreground, background, and border of a region. You might, for example, change the color of a key to show that it has been selected.

5.3.3.4 Specifying Color in Drawing Statements

You can draw different portions of a single object in different colors. For example, you might color the outline of a circle yellow, three lines of a triangle red, white, and blue, respectively, and a box solid magenta.

When you do not specify color in drawing statements, the drawing statements inherit the foreground color defined for the object. But if you do specify a color in a drawing statement, that color takes precedence over the color specified for the object. For example, consider the following object definition:

```
yellow key Box at position 50 50
:
    blue box 25 25
    move to 100 100
    circle radius 20
```

The Box key has the foreground color yellow. The circle is drawn in yellow, because no color is explicitly specified for it. The box is drawn in blue, because the **box** drawing statement specifies **blue**, which overrides the color of the object (yellow).

5.3.4 Ancestry

The ancestry attribute specifies the parent that contains the object. In fullscreen applications, the parent specified can be **master**; in windowed applications, the parent specified can be **desktop**, indicating that the object should be placed outside the primary region. If ancestry is specified for an object, that object is called the *child* of its *parent* object. The only objects that can be parents are graphical regions, image regions, dialog regions, and dialog boxes.

Action statements that change the parent object, except the **change graphics** or the **clear graphics** statements, also change all of its children. For example, if a parent is made invisible, all of its children will become invisible; if it is moved, all of its children move along with it. For example:

```
graphical region Panel size 300 200 at position 100 100  
    in ControlDisplay
```

```
key Button1 at position 50 50 in Panel in ControlDisplay
```

Here, Button1 is the child of Panel, and Panel is the child of ControlDisplay. Another way of saying this is that ControlDisplay is the parent of Panel, and Panel is the parent of Button1.

If you specify another key in Panel:

```
key Button2 at position 75 75 in Panel in ControlDisplay
```

Button1 and Button2 are *siblings*. If ControlDisplay is changed in size or position or in any other way, both Button1 and Button2 are similarly affected.

If you do not specify ancestry in an object definition, then the ancestor is by default the primary region or, for fullscreen applications, **master**.

See section 5.4.3, "The Ancestor Coordinate System" on page 5-20 for more information on the ancestry coordinate systems.

5.3.4.1 Valid Ancestors of Objects

The following object types and associated valid parent types are the only ancestry combinations that can be specified.

Object Type	Valid Parent Types
dialog box	desktop
Dialog Controls	dialog region and dialog box
dialog region	desktop , graphical region , and image region
graphical region	desktop , graphical region , and image region
image region	desktop , graphical region , and image region
key	graphical region and image region
sense region	graphical region and image region
textual region	desktop , graphical region , and image region

5.3.4.2 Specifying Ancestry in Object References

More than one object can have the same identifier, as long as the objects have different ancestry. For example, you might want to define two buttons, both named Button1. One is the child of Panel, and the other is the child of NewPanel. You might specify:

```
key Button1 at position 50 50 in Panel
  box 20 20
key Button1 at position 100 10 in NewPanel
  box 50 50
```

Thereafter, whenever one of the objects named Button1 is referenced in your EASEL OS/2 EE program, you must specify a name containing enough of the object's ancestry to distinguish it unambiguously from other objects; for example:

```
make Button1 in Panel invisible
```

```
make Button1 in NewPanel invisible
```

But, if you have defined only one Button1, you can specify:

```
make Button1 invisible
```

without specifying its ancestry.

Be careful that you do not accidentally delete objects that have children. If a parent object is deleted, you cannot reference any of its children, because when a parent is deleted, so are its children.

EASEL OS/2 EE can convert an object name with ancestry, such as PanicButton in Panel, to a string.

You can change the ancestry of objects during runtime (see 6.3.5.3, “Changing Ancestry in the change position to Statement” on page 6-33).

5.3.5 Priority

Two or more objects can be drawn at the same place on the display. When one object is the child of another object, the child is always drawn on top of its ancestor. When two objects are siblings, however, either because they are defined as offspring of the same parent object or because they are children of the screen, then it is the **priority** of the objects that determines which is drawn on top of the other. An object with a higher priority will be drawn on top of an object with a lower priority. If two siblings have the same priority, the last one added or modified will be on top. Therefore you should always assign appropriate priority numbers to objects that overlay one another.

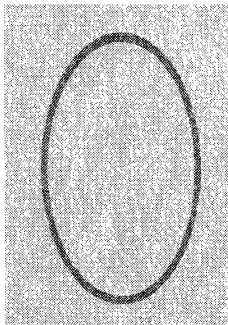
A priority number can be specified as any integer value: the larger the number, the higher the priority. If you do not assign a priority, the default priority of 0 is used for the object. In the key definitions below:

```
red key RedBox at position 200 100
  solid box 400 400
```

```
blue key BlueEllipse at position 400 300
  priority is 2
  ellipse major 100 minor 300
```

BlueEllipse is assigned a priority of 2; RedBox has the default priority of 0. BlueEllipse is therefore on top of RedBox.

These objects are displayed as shown below:

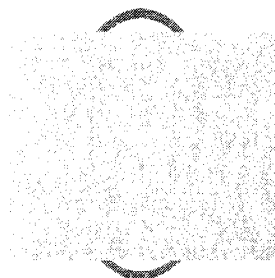


What you see on the screen is RedBox beneath BlueEllipse. Both objects have the attribute of visibility, but the lower priority of RedBox places it beneath BlueEllipse.

You can change the priority of objects during runtime with the **change priority** action statement. Continuing with the example shown above, if the priority of RedBox is changed so that it has a higher priority than BlueEllipse:

change RedBox priority to 3

RedBox is on top of BlueEllipse, as shown below.



What is now displayed is the entire box, with the ends of the ellipse extending above and below it.

5.3.5.1 Using Priority

A common place to use priority is with sense regions. Sense regions are often used to allow selection of a new position for an object. Because a sense region has no visual contents, it does not obscure other objects. Unless you give it a priority higher than other objects, it remains behind them and EASEL OS/2 EE takes responses to them rather than to the sense region when repositioning objects.

5.3.6 Parameter

The parameter attribute allows you to specify a value that can later be examined. It is returned by the built-in functions **parameter** or **parameter of**. This attribute does not affect the appearance of the object.

If you specify a parameter in an object definition, you can then test the value returned by the **parameter** or **parameter of** built-in functions, and perform action statements based upon this value. The **parameter** attribute, when used with the **parameter** built-in function, is especially useful when specifying actions for individual objects in a class. See 6.4.8.3, "The parameter Function" on page 6-40.

5.4 Coordinate Systems and Cursors

The *monitor* is the physical device on which your display is shown, and the *display* is an exact portrayal, in bits, of the image that is to appear on the monitor. The EASEL OS/2 EE *screen* is a logical coordinate space in which objects are arranged. It is portrayed on a display, which is then shown on the monitor.

5.4.1 The Screen Coordinate System

The **screen size** environment declaration defines the number of horizontal and vertical coordinate positions into which the screen is divided. This statement is optional. If you do not specify it in a windowed application, the screen defaults to the resolution of the display; in a fullscreen application, the screen defaults to a resolution of 800 600.

If you define a screen size, you must do so at the beginning of the program. You cannot define more than one screen size per program, and the size, whether specifically defined or defined by default, cannot be changed during runtime.

Model

```
screen size { X Y | dialog units | device units }
```

You can divide a screen into any number of logical horizontal and vertical positions. Once the screen size has been established, it can be thought of as a two-dimensional grid of X (horizontal) and Y (vertical) coordinates in which all objects are placed. Object positions (every object must have a specified position) are within these coordinates. Any objects positioned outside of the coordinates specified by the **screen size** statement will not be displayed.

You can specify virtually any screen coordinate system that is appropriate for the objects that you are defining. Typically, the screen size should equal the resolution of the computer's graphics display adapter, which you can access by specifying the following:

```
screen size device units
```

Once you have defined a screen size, you can run the EASEL OS/2 EE program on any graphics display adapter or resolution, and the screen coordinate system will get mapped to the actual resolution of the new display. You do not need to change the defined screen size. For example, if a program is written on a computer using an EGA and you define a resol-

ution of 640 350, and run the program on a computer using a completely different graphics board or resolution, all objects will be mapped proportionally to the new resolution, even without specifying a new screen size.

The only advantage to defining a different screen size than the current one is if the current resolution has very poor proportions.

A monitor is usually 3/4 as high as it is wide. In order to preserve the proportions of circles, arcs, and ellipses, it is recommended that you preserve these proportions in defining new screens. If you define a screen such as the following:

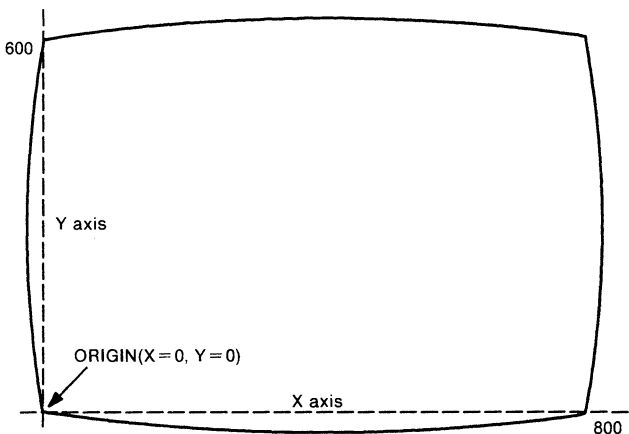
```
screen size 100 300
```

and display a circle on that screen, the circle is not displayed as exactly round, because the screen's proportions are not 4:3.

You can also define a screen size to use dialog units. Dialog units are proportional to the size of the PM system font. A dialog unit is defined to be one-quarter of the average width and one-eighth of the maximum height of the system font.

This is especially useful for applications that will run on different resolutions. If you use the system font in these applications, the application will appear with the same proportions on monitors with different resolutions.

The *origin* of the EASEL OS/2 EE screen coordinate system is located at the lower left corner of the display, as shown below. This position is always 0 0 in the screen.



The screen size defines the coordinate system of the screen, which is also the size of the **desktop** in windowed applications and **master** in fullscreen applications.

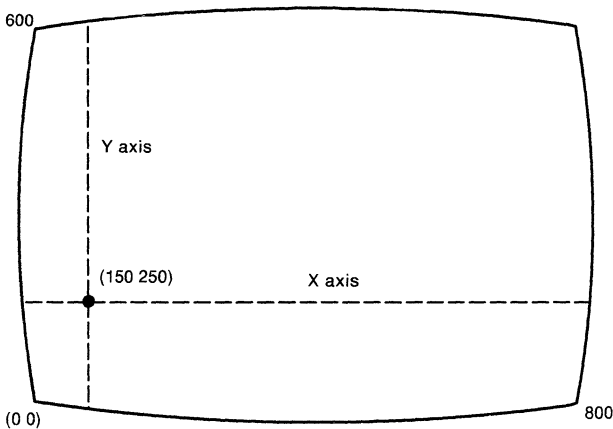
5.4.2 The Object Coordinate System

Every object has its own coordinate system, and all positions inside an object are specified using the coordinate system of that object. The object's *origin*, or starting position, is specified by the X coordinate (its position from left to right in a horizontal direction) and the Y coordinate (its position from bottom to top in a vertical direction) in the coordinate system of its parent.

The starting point for that object's own coordinate system is 0 0. This is the position at which EASEL OS/2 EE starts drawing the object. For example, the object definition:

```
key Logo at position 150 250
```

specifies a starting position (origin) for Logo as the point that is 150 positions in the X direction and 250 positions in the Y direction, measured from 0 0 at the lower left corner of the screen. Position 150 250 in the coordinate system of the EASEL OS/2 EE screen is equivalent to position 0 0, the origin of Logo, in the coordinate system of Logo. The origin of Logo is shown below:

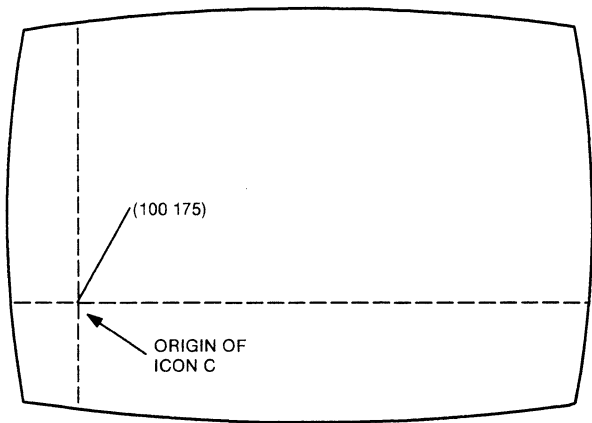


5.4.2.1 Drawing in the Coordinate System of Graphical Objects

Any drawing statements that are part of an object definition are executed relative to the object's origin. For graphical objects, the drawing statements are specified in X and Y coordinates. For example:

```
key C at position 150 250
  draw by 100 175
```

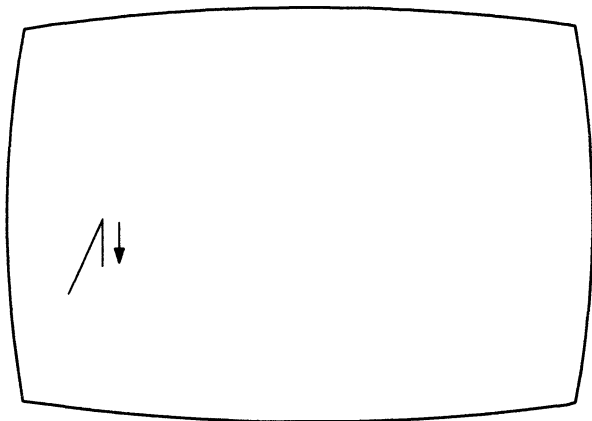
The image that would appear on the screen is:



You can specify negative numbers as coordinate positions. Suppose that EASEL OS/2 EE next executes the following **draw** statement:

`draw 0 -100`

EASEL OS/2 EE draws the following line:



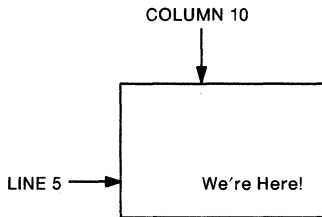
5.4.2.2 Drawing in the Coordinate System of Textual Regions

Like keys and graphical regions, any drawing statements that are part of the definition of a textual region are executed relative to the textual region's origin. But the coordinate system of a textual region is based on lines and columns, not in X and Y coordinates. A position in a textual region is addressed in terms of the column number and line number of the desired point. The top line is line 1, and the leftmost column is column 1 (zero and negative column or line numbers are not allowed).

For example, suppose you specify:

```
textual region News window size 50 columns 50 lines
  at position 100 100
  move to column 10 line 5
  insert "We're Here!"
```

EASEL OS/2 EE inserts the words "We're Here!" beginning at column 10, line 5 of the textual region, as shown below:



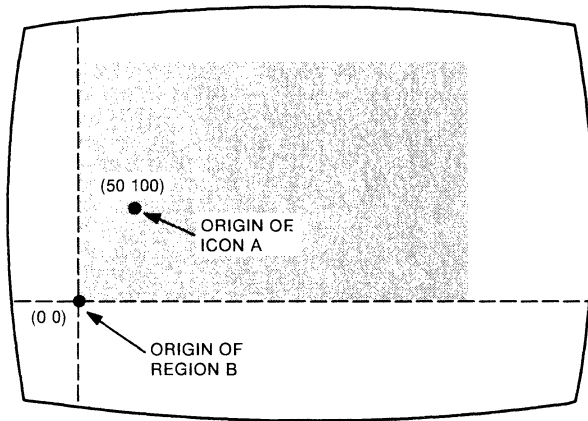
5.4.3 The Ancestor Coordinate System

The position of any object is in the coordinate space of the object's ancestor. Suppose you define an object as a child of a graphical region, as follows:

```
graphical region B size 400 300 at position 100 100

red key A at position 50 100 in B
```

The graphical region B is the parent (ancestor) of the key A. The **at [position]** specification for A is specified in the coordinate system of B, as shown below.



A textual region cannot be the ancestor of any object.

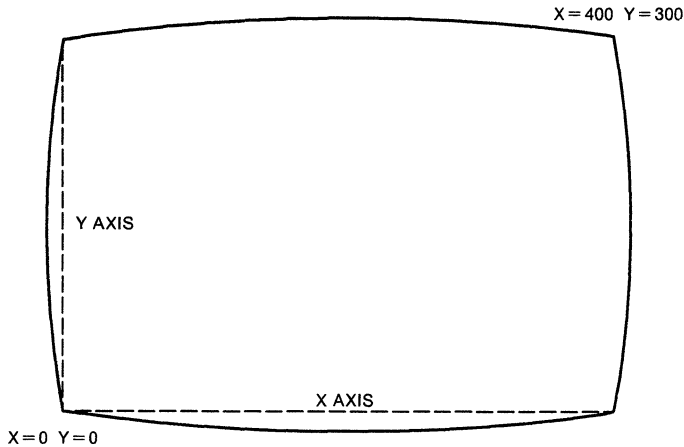
5.4.3.1 The Screen as Ancestor

When you define an object, you can specify its ancestry; that is, you can indicate that the object is the child of another object, or of the screen itself. If you define an object without specifying an ancestor (parent), its ancestor by default is the primary region in windowed applications; **master** (the screen) in fullscreen applications. To define an object as a child of the screen itself in windowed applications, specify **desktop** as its parent.

You can explicitly define the screen size, giving it any coordinates that you wish, or you can allow it to be defined by default. The position of the object is specified using the coordinate system of the screen. Suppose you specify the following **screen size** statement in an EASEL OS/2 EE program:

```
screen size 400 300
```

The screen has the coordinates shown below.

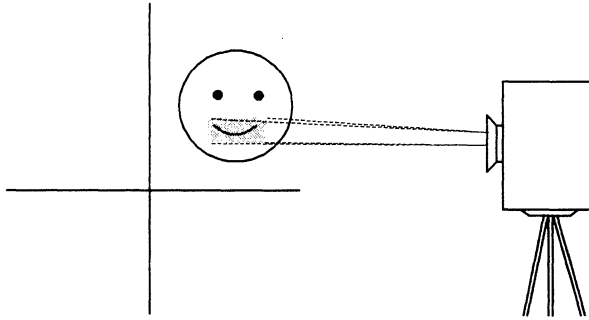


5.4.4 Windows and Viewports: The Coordinate Systems of Regions

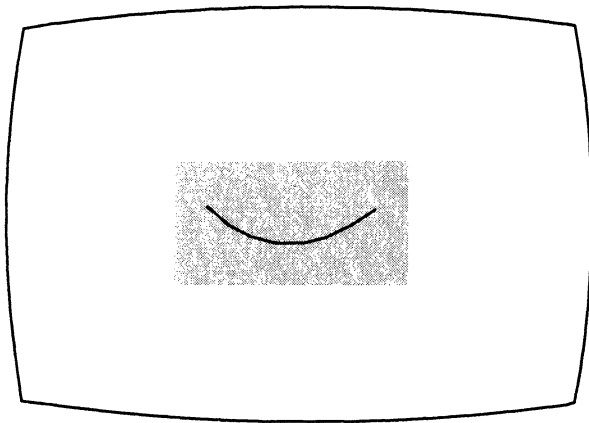
When you define a region, the region itself is an independent entity, not a part of the screen. The region definition actually defines three distinct characteristics:

- The region's *world*, which, similar to keys, consists of all the contents of the region. The world's coordinate system is infinite in all directions; drawing statements and children can be placed anywhere in the coordinate space.
- A *viewport*, with which you select the rectangular portion of the screen in which the contents of the window are visible.
- The region's *window*, the rectangular portion of the region that you wish to show in the EASEL OS/2 EE screen. The window is the area in the coordinate system of the world that is displayed through the viewport in the EASEL OS/2 EE screen.

Displaying the contents of a region's window in the region's viewport is like using a camera. You control what is visible in the camera's lens by aiming the camera. What you see through the lens is like the window of a region, as shown in the example below. (The diagrams in this section are shaded to illustrate the window or the viewport of a region; the shading is not actually displayed.)



You can take a snapshot of the window. The viewport is like the snapshot of the window. You can then paste the snapshot into a photograph album. The album is like the EASEL OS/2 EE screen; it is the place where you display the viewport. The image seen in the viewport shown above would look like the one shown below:



In the region definitions, the **size** and **at [position]** specifications define the initial size and position of the region. The **at [position]** specification defines the lower left corner of the region's viewport. The **size** specification defines the initial amount of space the region's viewport will take up in its parent's coordinate system.

The region's window is by default initialized as an area the same size as the viewport. Thus, the **size** specification also specifies the amount of space the region's window will display in its *own* coordinate system.

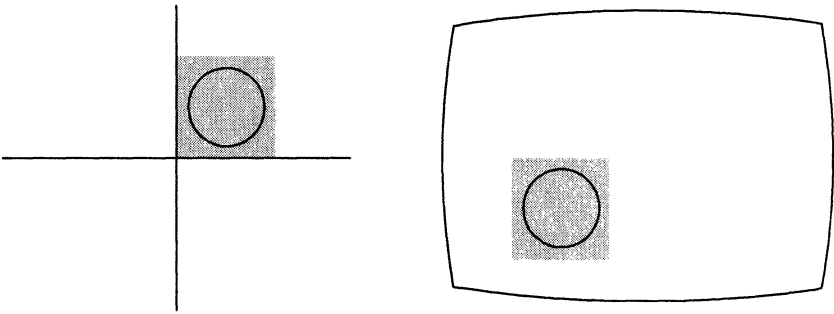
You can also use the **window size** attribute specification to explicitly specify a window size and position different from those of the viewport.

In the following graphical region definition:

```
graphical region Template size 300 300
  at position 100 100
  move to 150 70
  circle radius 50
```

the viewport for this graphical region is a rectangular area 300 coordinate positions square, located at position 100 100 of the screen. The window for this graphical region is also 300 coordinate positions square, and its lower left corner is position 0 0 in the graphical region. Any contents that occupy that area of the graphical region are in the graphical region's window.

In the figure below, the diagram on the left shows the graphical region, its contents (a circle), and the position of its window. The graphical region's viewport, as displayed on the screen, is shown on the right.



The circle is entirely within Template's window, and is therefore entirely portrayed in the viewport.

The size of a region's viewport is specified in the coordinate system of its parent. A region itself is not part of the coordinate space of its parent. It is only the region's viewport that is in the coordinate space of its parent.

A region has exactly one viewport and one window. Although you can change the size and position of either the viewport or the window, there are never more than one of each.

5.4.4.1 Windows

The contents of a region are portrayed in the viewport only if they are within the region's window. Initially, the contents of the window are displayed at their full size in the region's viewport. A region's window is the only portion of the region that can be seen; the contents of a region that lie outside the region's window are not displayed in the viewport.

Once you have defined a textual region, the window size and/or position can be changed by specifying the **change window size** or the **change window position** action statements in a response definition or in an action subroutine definition. You can control what is visible in a region's window by changing the window's position or dimensions. As with a camera, if you zoom in, less of the area is visible through the lens. If you zoom out, more of the area is visible through the lens. Similarly, you control the part of the region that is visible on the screen by changing the size or the position of the window. You can:

- Reduce the **window size** to display less of the region (zoom in).
- Increase the **window size** to display more of the region (zoom out).
- Change the **window position** to display a different part of the region (pan).

The following are the models for changing window size.

Model #1 (for Graphical, Image, Sense, and Dialog Regions):

```
change { REGION_NAME | REG_CLASS_NAME } window size
      [ by | to ] X Y
```

Model #2 (for Textual Regions):

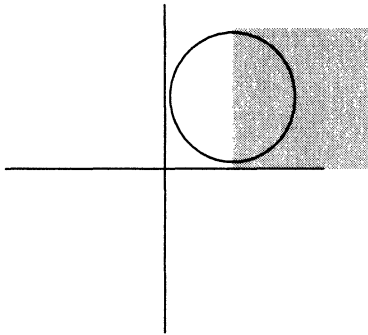
```
change { TR_NAME | TR_CLASS_NAME } window size
      [ by | to ] COLUMNS columns LINES lines
```

5.4.4.2 Changing Window Position

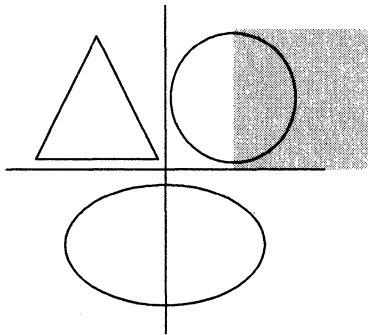
In the response definition below:

```
response to BigBox
  change Template window position by 150 0
```

In the response to selecting the object, BigBox, is to move the window of the graphical region, Template, 150 coordinate positions to the right. The Y position of the window does not change. Now, only the right half of the circle is in the window of this graphical region. The window and its contents are shown below:

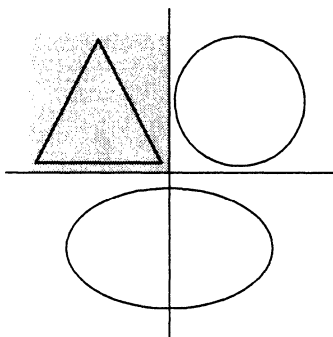


Suppose you add a triangle and an ellipse to the contents of Template, as shown below:

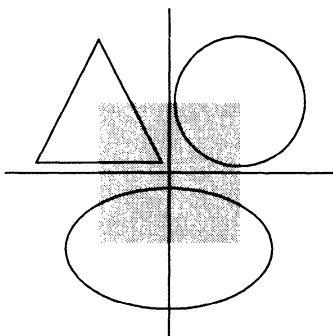


Only the right side of the circle is currently in the window. (The position of the window was changed in an action statement above.) The triangle, the ellipse, and the left side of the circle lie outside the bounds of the window, although they are still in the graphical region.

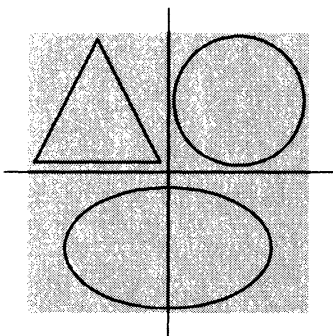
If you move the window to the left you can display just the triangle:



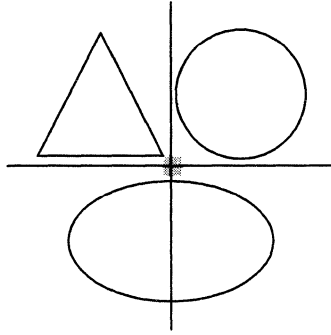
Now, the circle and the ellipse lie outside the bounds of the window. If you move the window downwards and to the right, part of each of the three shapes are in the window:



If you enlarge the window, all of the shapes are in the window:



If you reduce the size of the window, none of the shapes is in the window:



Both the size and the position of a window are entirely within your control. There are no practical horizontal or vertical boundaries, because a window is part of its limitless graphical region.

5.4.4.3 Viewports

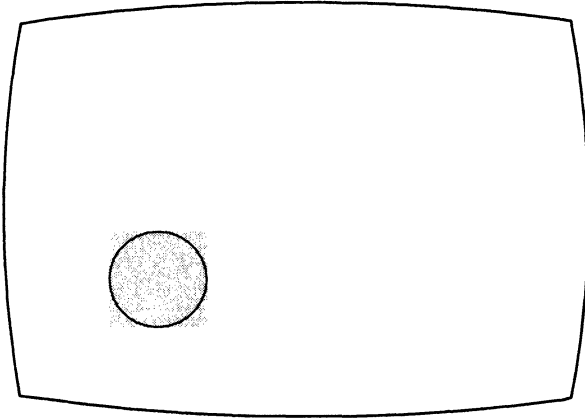
The viewport is a rectangular area of the region's ancestor on which the contents of the region's window are portrayed. Once you have defined a region, the size or the position of a viewport can be changed by specifying the **change size** or the **change position** statements as action statements in a response definition or in an action subroutine definition. You can control the way in which the contents of a region's window are displayed in the viewport by changing the viewport's position or dimensions. You can:

- Move the viewport to display the contents of the window on a different part of your screen.
- Change the size of the viewport to make the contents of the window appear smaller or larger.

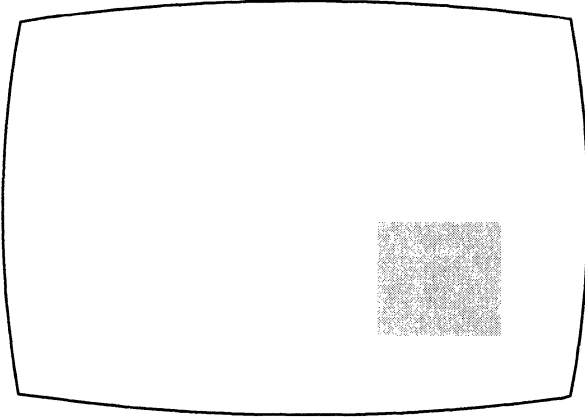
For example, the Template graphical region defined earlier:

```
graphical region Template size 300 300  
  at position 100 100
```

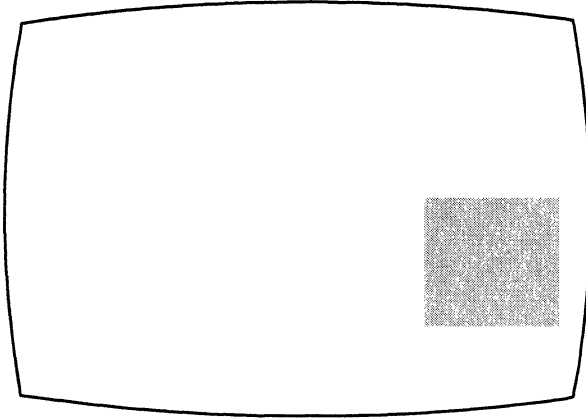
creates a graphical region whose viewport is 300 positions square beginning at position 100 100 of the EASEL OS/2 EE screen. Initially, only the circle is in the window of this graphical region. The graphical region's viewport is shown below:



You can use a **change position** statement to move the viewport to the right side of the screen, creating a display as shown below:



You can use a **change size** statement to change the size of the viewport. In the example below, the viewport for Template is bigger, so the circle is displayed larger than it actually is:



You can make the viewport larger, smaller, taller and narrower, or shorter and wider.

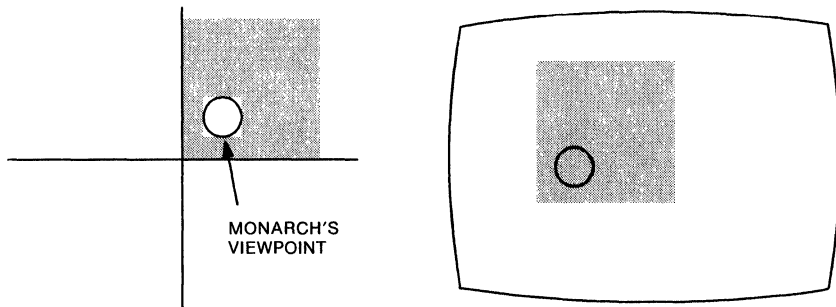
5.4.4.4 Effects of Ancestry in Windows and Viewports

A graphical or image region can be the parent of another object; a textual region can be the child (but cannot be the parent) of another object. In the following graphical region definitions:

```
graphical region Butterfly size 400 400  
    at position 200 200
```

```
graphical region Monarch size 100 100  
    at position 50 50 in Butterfly  
    move 50 50  
    circle radius 50
```

Monarch is the child of Butterfly. The figure below shows Butterfly's window on the left, and Butterfly's viewport on the right:

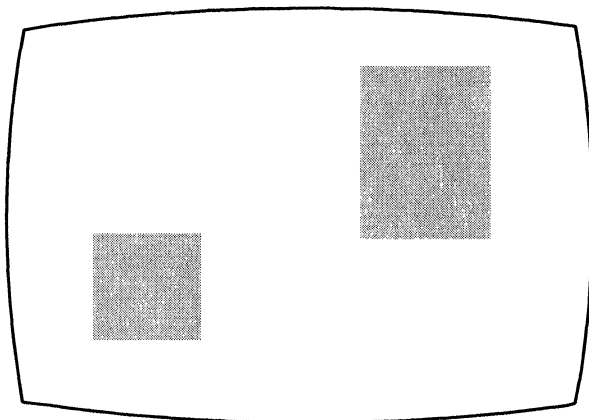


A region's viewport is part of its ancestor's coordinate space. Therefore, the viewport of the child, Monarch, is portrayed in Butterfly's viewport only as long as it is in the window of Butterfly. If you change the position of Butterfly's window so that Monarch's viewport is entirely outside of it, none of Monarch's viewport is portrayed on the screen. Similarly, if you move Butterfly's window so that only part of Monarch's viewport remains in it, only part is portrayed.

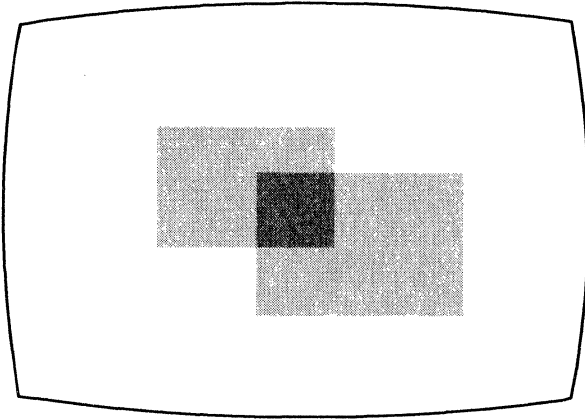
5.4.4.5 Multiple Regions

The viewports of many regions can be visible on the screen simultaneously. For example, you might show the viewport of one graphical region on most of the EASEL OS/2 EE screen.

In an action statement you can reduce the size of the viewport and change its position to show another graphical region's viewport at the same time, for example:



You can simultaneously show the viewports of two or more graphical regions at the same place on the screen. The viewports can be partially overlapping, such as those shown below:



or they can be the same size and occupy the same position.

It is important to remember that the a region's area is not limited to the area of your EASEL OS/2 EE screen. The display merely has a viewport onto the current contents of the region's window. You can create and manipulate graphical images anywhere within a graphical region without being constrained by what is visible in the EASEL OS/2 EE screen.

5.4.5 The Graphics and Text Cursors

Each key and graphical region you define has two cursors associated with it: a graphics cursor and a text cursor. A textual region has a text cursor but no graphics cursor. Sense regions and image regions have no cursors. The cursors determine the position at which drawing statements will start. When an object is first defined, both cursors are positioned at the object's origin. This position is updated each time one of the object's drawing statements is executed.

When a graphical drawing statement is executed, EASEL OS/2 EE first sets both cursors to the current position of the graphics cursor, then moves both cursors together to the position specified by the graphical drawing statement.

Every object has its own cursor(s). The cursor(s) in one object are not affected by drawing statements for any other object.

Neither the graphics cursor nor the text cursor is visible on the screen. When a cursor position is marked in the following examples, it is provided for purposes of illustration only.

5.4.5.1 The Graphics Cursor

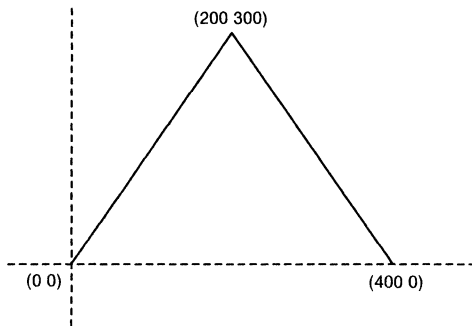
The graphics cursor is initially positioned at the object's origin—the point on the screen described by the **at [position]** specification in the object definition. Each time a drawing statement is executed, the position of the graphics cursor is updated.

The cursor position after the execution of a drawing statement depends on the statement executed. The cursor may be in a different place or in the same place. Some drawing statements, such as **box** and **circle**, begin and end at the same place, so the position of the graphics cursor is the same before and after the statement is executed. Other drawing statements, such as **draw** and **move**, change the position of the graphics cursor.

Suppose you define a key as follows:

```
key Arrow at position 100 200
    draw to 200 300
    draw to 400 0
```

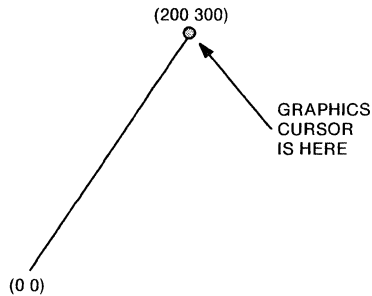
EASEL OS/2 EE displays the image shown below:



(The coordinates shown in this picture are Arrow's own coordinates. Arrow's origin is 100 200 in the screen coordinate system.) Before the drawing statements are executed, the initial position of the graphics cursor is at this origin. When EASEL OS/2 EE executes the first drawing statement:

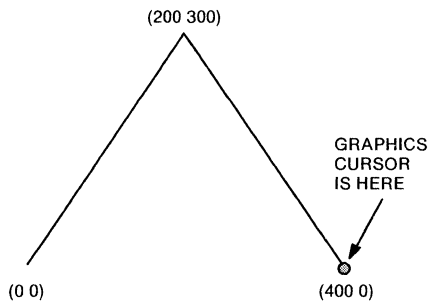
```
draw to 200 300
```

a line is drawn, as shown below:



The graphics cursor is now positioned at 200 300 in Arrow's coordinate system. When EASEL OS/2 EE executes the second drawing statement:
draw to 400 0

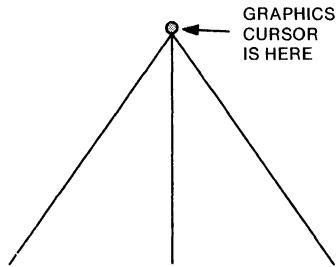
a second line is drawn, starting at the current position of Arrow's graphics cursor (200 300):



The graphics cursor is now positioned at 400 0 in Arrow's coordinate system. EASEL OS/2 EE retains the position of an object's graphics cursor, and any later additions to the contents of the object will begin at this point. For example, if the following response definition, which adds new drawing statements to Arrow's contents, is executed, the statements start at the current position of Arrow's graphics cursor (400 0), and draw a vertical line to the tip of the arrow:

```
response to MakeBigArrow
  add to Arrow
  move to 200 0      # Moves the graphics cursor.
  draw to 200 300    # Draws the line.
```

The final appearance of the object is shown below:



Arrow's graphics cursor remains at position 200 300, and any further changes to the contents of Arrow will begin at that point.

Some drawing statements allow you to specify *relative* or *absolute* points in the coordinate system. Relative positions, which are points relative to the current position, are specified with the **by** keyword; absolute positions are specified with the **to** keyword. For example, a drawing statement of the form:

```
draw by 100 100
```

draws a line whose endpoint is 100 positions to the right of the current position and 100 positions up from this starting position. Because this line is drawn relative to the current position, this statement is considered to be a relative drawing statement.

On the other hand, a drawing statement of the form:

```
draw to 100 100
```

draws a line whose endpoint is at coordinate position 100 100 in the object's coordinate space, regardless of the starting position of the line. Because EASEL OS/2 EE draws to this absolute position, this statement is considered to be an absolute drawing statement.

Examples of absolute and relative drawing statements are provided throughout this publication.

5.4.5.2 The Text Cursor

In addition to the graphics cursor described above, every object, including a textual region, has a text cursor associated with it. (A textual region has *only* a text cursor.) Just as the graphics cursor identifies the starting point for graphical drawing statements, the text cursor identifies the starting point for textual drawing statements. The presence of both a graphics cursor and a text cursor in keys and graphical regions allows you to intermix graphics and text operations easily in the contents of a single object.

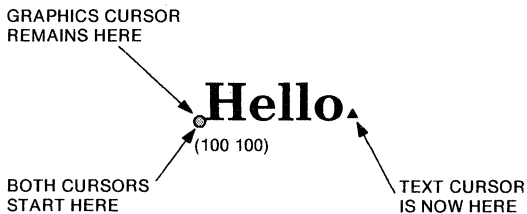
Usually, the graphics cursor and the text cursor are at the same place. Whenever the graphics cursor is moved, the text cursor is moved with it. EASEL OS/2 EE moves the cursors together because text is often used as a label for graphics. Therefore, when text is drawn it is usually drawn at the position at which the next graphics will be drawn. Drawing text does not move the graphics cursor, so when additional graphics are drawn these graphics will begin at the correct location.

Sometimes, however, text is not just a label for graphics. Several different pieces of text might be drawn, and it is necessary that each text start immediately after the previous text. Therefore, a separate text cursor is used to control where the text will be positioned. Consider the following examples of how the text cursor and the graphics cursor interact.

Suppose you specify the following:

```
key Begin at position 10 10
move to 100 100
text "Hello"
```

the text string is displayed as follows:

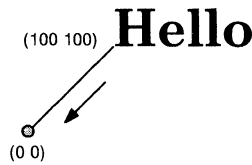


After the string is displayed, the graphics cursor remains positioned at 100 100, but the text cursor is positioned after the "Hello" string.

If the next drawing statement specifies a graphics operation, such as drawing a line, EASEL OS/2 EE begins at the graphics cursor position. The statements:

```
key Begin at position 10 10
move to 100 100
text "Hello"
draw to 0 0
```

cause the following to be displayed:



After these statements have been executed, both the graphics cursor and the text cursor are at coordinate position 0 0.

But the effect is different if you display successive text strings. The statements:

```
key Begin at position 10 10
move to 100 100
text "Hello"
text " Program User"
```

cause the following to be displayed:

Hello Program User.

After these statements have been executed, the graphics cursor is still at position 100 100, but the text cursor is at the position immediately following the "Program User" string.

After execution of a textual drawing statement, the text cursor is positioned at the lower left corner of the last drawn character.

5.4.5.3 Text Cursor in Textual Regions

A textual region is a place to display and manipulate lines of text; it cannot contain graphics. Therefore, a textual region has only one cursor, a text cursor. A textual region drawing statement begins at the current position of the text cursor, just as a drawing statement for a graphical object begins at the current position of the object's graphics cursor. But in textual regions, the text cursor is moved by line and column specifications, rather than by X and Y coordinates. For example, you can move the text cursor to a particular character position in a textual region by specifying a move statement such as the one shown below:

```
move to column 20 line 9
```

This statement moves the text cursor to the character that occupies the 20th position of the 9th line of the text in the textual region.

Although the text cursor is normally not visible on the screen, you can make it visible in a textual region. For details, see 8.4.5, “Changing the Visibility of the Text Cursor” on page 8-29.

5.5 Mixing Object Types in a Window

Displaying graphics, text, images, and dialog controls in the same object is not possible in EASEL OS/2 EE. However, displaying all these objects in the same window is possible in EASEL OS/2 EE. This can be accomplished by defining multiple object types as children of the same parent region. The child objects would be defined as having no frame attributes, unless a frame attribute is intended to be displayed to the user. The child object would also be defined with the same background color as the parent region, unless a different color is to be displayed to the user for the object. The parent region would then be defined as having the appropriate frame attributes for the window. By defining the child objects and parent region in this manner, the user will not be able to visually see that the window is made up of multiple object types.

The following is an example of this technique.

```
enabled visible color 26 primary graphical region PrimaryWindow
  size 400 300
  at position 40 30
  color 27 foreground
  size border
  title bar "Primary Window"
  system menu
  horizontal scroll bar scroll by 6
  vertical scroll bar scroll by 16
  no scale
  action bar PrimaryWindowActionBar
  minimize button using "minbuttn.ico"
  maximize button
```

```
enabled visible color 26 image region Image
  size 150 100
  at position 250 200
  in PrimaryWindow
  color 27 foreground
  no scale
  no cache
  file "picture.bmp"
```

```
enabled visible color 26 textual region Text
  size 250 150
  at position 0 150
  in PrimaryWindow
  color 27 foreground
  font medium
  insert file "profile.txt"
```

```
enabled visible dialog region ProfileUpdate
  size 300 100
  at position 0 0
  in PrimaryWindow
```

```
enabled visible entry field OldName
  size 60 11
  at position 50 60
  in ProfileUpdate
  text size 50 columns
  left align
```

```
enabled visible static text OldNamePrompt
  size 40 8
  at position 5 62
  in ProfileUpdate
  text "Old Name:"
```

```
enabled visible entry field NewName
  size 60 11
  at position 50 20
  in ProfileUpdate
  text size 50 columns
  left align
```

```
enabled visible static text NewNamePrompt
  size 40 8
  at position 5 22
  in ProfileUpdate
  text "New Name:"
```

```
enabled visible check box Married
  size 50 10
  at position 200 62
  in ProfileUpdate
  text "Married"
```

```
enabled visible check box Male
  size 50 10
  at position 200 22
  in ProfileUpdate
  text "Male"
```

When using this technique, the object which currently has the input focus can not be determined by the user, since the object does not have a title bar or size border frame attribute that changes color to indicate the current input focus. The parent region will have its title bar and size border high-

lighted, if they exist, when the parent region or one of its child objects has the input focus.

Care should be taken when defining windows in this manner when a keyboard will be used with the application. While one of the child objects has the current input focus, which cannot be determined by the user, the keyboard functions for the parent region, or another child object, may not be operational. However, all the keyboard functions for the region or object which does have the input focus are operational.

5.6 Fonts

5.6.1 Specifying a Font

The **font** statement specifies the font in which to display text. A font can be specified in the following places in the program:

- At the beginning of the program, as the default font for all text in the program. The font environment declaration should be placed just after the **screen size** environment declaration. For example:

```
screen size 640 480
font "asc59"
```

- If a defined font is used in the environment definition, its definition should be placed before the font environment declaration.

```
screen size 640 350
font Normal is facename "asc59" size 7 11 # font
                                           # definition
font Normal                               # font
                                           # env
                                           # declr
```

Note: When you declare a defined font, do not enclose the font name in quotation marks. For example, font Normal in the example above is not enclosed in quotation marks.

- In a drawing statement for an individual text string. For example:

```
add to StopSign
font Normal "STOP"
```

- In the definition of a textual region. For example:

```
textual region News
window size 19 columns 5 lines
:
at position 100 100
font "large"
```

You can reference up to 20 fonts, plus the default font (**medium**), in a program at compile time. If you want to reference more in a program, you must modify the FNTCNT option in your CONFIG.ESL file.

5.6.1.1 Inheritance of Font Type

The font in which a text string is displayed is inherited in the following ways:

- In graphical objects: Text that does not have a font specification is displayed in the default font for the program.
- In textual regions: Text is displayed in the font specified in the definition of the textual region. If the textual region definition does not specify a font, the text is displayed in the default font for the program.
- The default font for the program is one of the following:

Any font specified in the environment declaration for the program.

If no font is specified in the environment declaration, the default font **medium** is used. Thus, any text string that does not have an explicit font specification or that does not inherit a font is displayed in EASEL OS/2 EE's **medium** font.

5.6.2 Character Cells

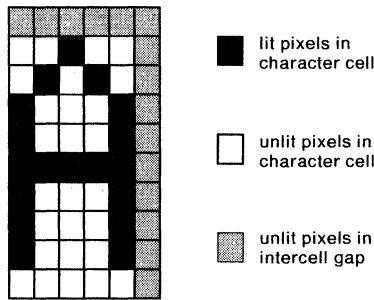
In order to display text effectively, it is important to understand how the characters that comprise a font are represented on a monitor.

Each character is displayed in a character cell. A character cell is a rectangular grid of pixels whose dimensions are font-dependent; a font defines the number of pixels in a character cell.

A *pixel* is an individual point on the monitor. A single character is displayed within a character cell by lighting the appropriate pixels. For example, font “**asc59**” has a character cell that is 5 pixels wide and 9 pixels high. Therefore, each character cell in this font has 45 pixels (5x9). When EASEL OS/2 EE displays a character, each pixel in the character cell is either on (lit) or off (not lit).

An *intercell gap* lies between any two character cells. The pixels in the intercell gap are always unlit so that there is space between adjacent letters, both for characters that are side-by-side and for characters that are on one line above another.

The figure below shows a total cell for a font with a character cell size of 5x9, and a total cell size of 6x10:

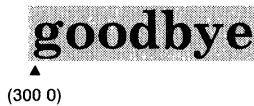


Certain characters have descenders. A *descender* is a part of a descending, lowercase character such as “g,” “p,” and “y.” The bottom of a character cell is used for descenders, as shown by the line of unlit pixels in the diagram of the letter “A” above. When you position a text string, the lowest point of the longest descender is positioned at the Y coordinate position. For example:

move to 300 0

text "goodbye"

is displayed as shown below:



5.6.3 Portability Across Monitors

Whereas graphics are displayed in screen coordinates (a property common across all monitors), text is usually displayed in pixels, a physical property of a particular display. A text string looks different on different display resolutions because a pixel represents a different percentage of each display screen. There are two categories of fonts:

In a **fixed-cell font**, a text string occupies the same number of pixels regardless of the resolution of the display.

In a **resolution-dependent font**, a text string occupies the same proportion of any display. One font name identifies a family of fonts, one for each display resolution.

In some cases, you might want to transfer a single EASEL OS/2 EE program from one display to another. Portability across displays can be achieved by using resolution-dependent fonts.

5.6.3.1 Fixed-Cell Fonts

A fixed-cell font allocates a certain number of pixels to each character cell. Regardless of the resolution of the monitor you are using, the same number of pixels is provided for each cell. This means that for a higher resolution monitor (960x720, for example), a text string appears smaller than for a lower resolution monitor (800x600). For example, a text string of 400 horizontal pixels requires one-half of the horizontal dimension of an 800x600 monitor, but 400 pixels represents less (about two-fifths) of the horizontal dimension of a monitor that is 960x720.

If you define a logical screen size of 800 600, a graphical or textual region that is 200 coordinate positions wide represents one-quarter of the monitor. On a monitor that is 960 pixels wide, the region is large enough to hold a text string that occupies 240 pixels. However, on a monitor that is 800 pixels wide, the same region would be too small to hold the text. Therefore, fixed-cell fonts should not be used when portability across monitor resolutions is required.

5.6.3.2 Resolution-Dependent Fonts

A resolution-dependent font is one in which each character cell occupies the same fraction of the monitor (subject to rounding differences that occur in the arithmetic), rather than the same fixed number of pixels. When you specify a font that is part of the family of resolution-dependent fonts (for example, the **medium** font), there is actually a font for each display resolution, and EASEL OS/2 EE displays the text in the font for the display that you are using. The text string therefore occupies the same proportion of the display, regardless of the resolution.

5.6.4 EASEL OS/2 EE Fonts

The following tables specify pixel sizes of EASEL OS/2 EE fonts. "Total size" includes the vertical and horizontal spacing used for each letter. These sizes will be true only if the screen size is the same size as the number of pixels in the display.

5.6.4.1 EASEL OS/2 EE Fixed Cell Fonts

Font Name	Total Size
asc37	4 x 9
asc59	7 x 11
asc79	9 x 12
asc510	5 x 10
asc713	7 x 13
asc88	8 x 8
asc814	8 x 14
caps814	8 x 14
asc1115	13 x 18
asc1521	17 x 24
medvt10	8 x 10
medvt12	8 x 12
medvt14	8 x 14
medvt16	6 x 16
medvt20	8 x 20
medvt24	8 x 24

5.6.4.2 EASEL OS/2 EE Resolution-Dependent Fonts

Resolution	Font Name	Total Size
640x350	medium	8 x 8
	large	12 x 14
	mcaps	8 x 8
	lcaps	12 x 14
640x480	medium	8 x 12
	large	712 x 19
	mcaps	8 x 12
	lcaps	12 x 19
1024x768	medium	12 x 18
	large	19 x 30
	mcaps	12 x 18
	lcaps	19 x 30

5.6.4.3 Converting Text Strings into Uppercase Characters

The EASEL OS/2 EE fonts **mcaps**, **lcaps**, and **caps814** are used to *convert* and display a text string in all uppercase characters. For example:

disabled key Sample at position 10 10
red font "lcaps" "I am all large caps"

The **mcaps** font displays text in the same size as the **medium** font; the **lcaps** font displays text in the same size as the **large** font.

5.6.5 Presentation Manager Fonts

All PM fonts are available in EASEL OS/2 EE. The PM fonts, unlike the EASEL OS/2 EE fonts described above, can be proportional.

5.6.5.1 Fixed-Width and Proportional Fonts

In a fixed-width (monospaced) font, every character is the same width. For example, the asc814 character cell is 8 cells wide and 14 cells high and a thin letter like “i” takes up the same space as a wider letter like “m.” All the EASEL OS/2 EE fonts are fixed-width.

In a proportional font, a thin letter like “i” takes up fewer pixels than a wider letter like “m.” *Helv*, *Tms Rmn*, and *System Proportional* are proportional PM fonts. *Courier* is a fixed-width PM font.

Note: Do not confuse the width characteristic of a font with its resolution dependence, discussed above. All the fixed-cell and resolution-dependent fonts described above are fixed-width.

5.6.5.2 OS/2 Presentation Manager Fonts

The following table shows the currently available PM fonts and their character-cell sizes.

Facename	Type	Available Sizes
Courier	Fixed-width	8x7, 8x10, 8x13 9x8, 9x12, 9x16 10x10 12x10, 12x12, 12x15, 12x20 15x15
Helv	Proportional	5x6, 5x10, 5x13 6x8, 6x10, 6x12, 6x16 7x10, 7x12, 7x15, 7x20 8x12, 8x18, 8x24 9x15 10x18, 10x22 11x15, 11x29 13x22 14x19, 14x28, 14x37 18x28
System Proportional	Proportional	Depends on display resolution. Specify a size of 0-0.
System Monospaced	Fixed-width	8x8, 8x12, 8x16 9x20

Facename	Type	Available Sizes
Tms Rmn	Proportional	4x6, 4x10, 4x13 5x12, 5x16 6x8, 6x10, 6x15, 6x19 7x10, 7x11, 7x12, 7x16, 7x21 9x15 10x14, 10x16, 10x20, 10x27 13x21, 13x26, 13x35 14x18 27x27

Enter font "system" if you want to use the system proportional font without an attribute (bold, italic, underline, strikeouts, or outline) in a program. In order to use any other PM font, you must define it using the **font is** statement.

5.6.6 Defining a Font

A font is defined by using **font is** statement shown in the following model.

Model

```
font FONT_NAME is facename FACE_NAME
    size XSIZE YSIZE
    [bold] [italic] [underline] [strikeout] [outline]
```

Suppose you want to use the Helv font, size 9x12, to display text in your program. First define it as follows:

```
font TextFont is facename "Helv" size 9 12
```

Note: Type the facename in uppercase or lowercase exactly as it is shown in the tables; facename helv is not the same as facename Helv.

When the font is defined, you can display text by referencing the font name as follows:

```
font TextFont text "Select the blue button when ready."
```

Because Helv is a proportional font, the text "Select the blue button when ready." will appear on the screen in letters that are 12 pixels high and, on average, 9 pixels wide. The "l" in "blue" will be less than 9 pixels wide; the "w" in "when" more.

You can define a font with **bold**, **italic**, **strikeout**, and **underline** attributes. You might choose to underline the word "blue" to emphasize that the user should select the blue, not the red, button. To do this, you must define a second font, which is exactly like the first except that it is underlined:

```
font UFont is facename "Helv" size 9 12 underline
```

Then change the actions that display the text as follows:

```
font TextFont      text "Select the "  
font UFont          text "blue"  
font TextFont      text " button when ready."
```

You can specify these attributes for all the PM and EASEL OS/2 EE fonts. You could also display the same text in the EASEL OS/2 EE font **medium**, by defining an underlined version of font **medium** as follows:

```
font MLine is facename "medium" size 8 12 underline
```

Note: Type **medium** and the other EASEL OS/2 EE facenames in uppercase or lowercase exactly as shown in the table of EASEL OS/2 EE resolution-dependent fonts, above. The size given in the example for font **medium**, 8x12, (for screen resolution 640 x 480) was obtained from this table.

If "medium" is the default font, the following actions will display the text as desired.

```
          text "Select the "  
font MLine text "blue"  
          text " button when ready."
```

To use the System Proportional font in a **font is** statement, specify a size of 0 0. The appropriate size will be determined at runtime.

```
font SystemItalic is  
  facename "System Proportional"  
  size 0 0  
  italic
```

5.6.7 Image Fonts

The fonts described so far are not scalable; all have a known character-cell size. On a given monitor, text displayed in these fonts will occupy a given area in pixels. Even if the graphical region containing the text is scaled, the text continues to occupy the same area in pixels. Fonts used to draw nonscalable text are called bit-map or image fonts.

5.6.8 Scalable Outline Fonts

You can make most PM fonts scalable by specifying the **outline** keyword when you define them. Outline fonts are not represented by character-cell maps. Instead, each character in an outline font is a series of graphical drawing commands (similar to the **shape** statement). Because of this, outline font text behaves exactly like any other graphical object. When its region changes shape, it scales accordingly. Here is a sample definition of an outline font:

```
font Scalable is facename "Courier" size 60 60 outline
```

Each character drawn in font Scalable occupies approximately 60 units of height and 60 units of width in the coordinate system of the object it is drawn in. If you draw a 60 by 60 box at the same point as a character in font Scalable, the character always appears in the box no matter how you scale the object because Courier is a fixed-width font. If you specify a proportional facename in place of Courier, the 60 by 60 box is the width of an average character, and wider letters overlap it.

The height of an outline font is the distance from the baseline (the bottom of a character, not including its descender) of the letters to the top of the highest letter.

Image fonts are drawn much faster than outline fonts because more calculations are required to draw and fill the shapes of an outline font.

Large outline text looks good but small outline text may be ragged because of rounding errors. It is therefore not as readable as small image text.

When deciding which font to use, you should consider whether speed is important, what size the text will be, and whether it needs to be scalable. You may wish to try out the fonts in a small program before deciding which one is best in each instance.

5.6.9 Fonts in Textual Regions

You can specify the **font** attribute in a textual region definition. A textual region can have just one font, which must be fixed-width (not proportional) and image (not outline). Fonts in textual region cannot be defined with any of the following attributes: bold, underline, strikeouts, or italic. See section 5.6, "Fonts" on page 5-42 for more information on fonts.

If you specify a font, all text in the textual region will be displayed in the specified font. If no font is specified, the text will be displayed in the default font for the program. If no default font is specified for the program, the default font of **medium** will be used.

You can change the font in a key or graphical region during runtime by specifying the new font in a drawing statement within a **change** action statement.

PM fonts are allowable in textual regions with certain restrictions. Textual regions are by nature unscalable. Each character must occupy the same amount of space. Therefore, only fixed-width image fonts are allowed in textual regions. The **underline**, **bold**, **italic**, and **strikeout** attributes are not allowed in textual regions. If you specify a font that is not allowed, it is replaced by the default font.

Note: If the default font of your program is proportional or scalable, font **medium** is used in all textual regions.

5.7 Redrawing an Object

This section explains how and when to redraw an object. Two action statements are provided for redrawing an object:

Statement	Purpose
refresh	Redraws a specified object.
wait 0	Causes EASEL OS/2 EE to immediately redraw those objects that were changed by previous statements in the current response definition.

5.7.1 Redrawing a Specified Object

The **refresh** action statement affects only one specified object at a time. The specified object is completely erased and then redrawn.

Model

refresh OBJECT_NAME

Like other action statements, **refresh** can be specified in a response definition or in an action routine definition.

The **refresh** statement may slow down your program, and therefore should be used only when you find that there is a discrepancy in the screen display—for example, if part of an object continues to show through another overlapping object, when it should be obscured by the overlapping object.

Chapter 6. Responses, Blocks, Action Statements, and Classes

6.1 About Responses

6.1.1 Overview of Responses and Their Uses

A response definition consists of two kinds of statements: a *response statement*, which specifies the stimulus, or event, that is to invoke a response; and *action statements*, which specify the actions that EASEL OS/2 EE is to perform when a particular stimulus is recognized.¹ For example:

```
response to MakeRed          # response statement
make BigBox red              # action statement
make BigBox visible          # action statement
send "Red \n" to Recorder    # action statement
```

Each response statement specifies a stimulus to which EASEL OS/2 EE can respond. You specify a response statement as the first statement in a response definition. Following the response statement, you can specify any number of action statements. The complete set of response statements are as follows:

response to start: You specify this statement in a response definition that is to be invoked when the EASEL OS/2 EE program is initiated, or when a block is first entered.

[interrupt] response to OBJECT: You specify this statement in a response definition that is to be invoked when the user selects an object.

[interrupt] response to item in OBJECT: You specify this statement in a response definition that is to be invoked when the user selects an action bar item (optionally in an object).

[interrupt] response to line or **[interrupt] response to char:** You specify either of these statements in a response definition that is to be invoked when the program receives data from the keyboard or from an application program.

[interrupt] response to stimulus: You specify this statement in a response definition that is to be invoked when a stimulus library generates an event.

¹ You can specify a response statement with zero action statements; for example, to clear the buffer in response to certain characters you know will be sent from an application; or, when creating a prototype interface, you may have a response to an object which eventually will send a string to an application.

response to low memory: You specify this statement as a response definition that is to be invoked when EASEL OS/2 EE reaches a specified threshold of available memory during runtime.

[interrupt] response to interval: You specify this statement in a response definition that is to be invoked at specified intervals (in seconds) during runtime.

[interrupt] response to timeout: You specify this statement in a response definition that is to be invoked when EASEL OS/2 EE does not receive any stimulus for a specified number of seconds.

response to termination: You specify this statement in a response definition that is to be invoked when a block is left implicitly.

In summary, EASEL OS/2 EE responds to the following stimuli:

- Selecting an object on the screen (with any pointing device).
- Selecting an item from an action bar.
- Entering data on the keyboard.
- Receiving data from an application program.
- Events generated by a stimulus dynamic link library.
- Initiation of the EASEL OS/2 EE program, or of a defined block of responses.
- Elapsed time.
- Lack of any stimuli during elapsed time.
- When a block is left implicitly.
- Specified pointing device events.

6.1.2 Responding to Program Startup

You can specify a **response to start** statement that tells EASEL OS/2 EE what to do when that program first begins execution. This kind of response is helpful if you want to perform operations such as starting an application program, logging into a remote computer, or displaying a menu of program options at the beginning of the program, before EASEL OS/2 EE performs any other operations. For example:

```
response to start
  make FirstMenu visible
  start local Audit "audit"
  send StartupMessage to Audit
```

EASEL OS/2 EE executes these statements as soon as the program begins, or when another EASEL OS/2 EE program transfers control to this program (with the **change to program** action statement.)

The **response to start** definition is optional. Only one **response to start** is allowed in an EASEL OS/2 EE program, unless it is within a block (and there can be only one **response to start** within each block).

6.1.3 Responding to Objects

You specify the **response to OBJECT_NAME** in a response definition that is to be invoked when the user selects an object. (You can also specify the **response to CLASS_NAME** statement in a response definition that is to be invoked when the user selects any member of a class of objects.) When the object is selected, EASEL OS/2 EE responds by executing the action statements specified in the response definition. The response will be invoked only if the object is selectable (**enabled**) at the time that it is selected.

6.1.3.1 Response to Object Selection

Use the following model to respond to the user's selection of an object, or of any object in a class.

Model

```
[interrupt] response to { OBJECT_NAME | CLASS_NAME }  
  
[ACTION_STATEMENT]...
```

The **interrupt** keyword is described in 6.2.8, "Responding to Specified Stimuli and Returning to a Block" on page 6-19.

6.1.3.2 Response to Other User-Generated Selections

Use the following model to respond to the user's selection of an object, or of any object in a class, if the user:

- Activates the region (**on activate**).
- Selects the object by clicking once with the primary mouse selection button or by using keyboard (**on button1 down**). This is the same as using the "response to object selection" model.
- Double-clicks on the object with the primary mouse selection button (**on button1 double-click**).
- Selects the "Close" choice from the system menu, or double-clicks on the system menu (**on close**).

A response may contain one or more of the **on** clauses in any order. However, a given response may contain no more than one of each type. For example, only one **on close** clause is allowed in any one response.


```
[interrupt] response to { OBJECT_NAME | CLASS_NAME }  
  on { activate | button1 down | button1 double click | close }  
  
    [ACTION_STATEMENT] ...  
  
  [on { activate | button1 down | button1 double click | close }  
  
    [ACTION_STATEMENT] ... ] ...
```

An **on button1 double click** event is always preceded by an **on button1 down** event. The corresponding actions for **on button1 down** (or the generic **response to ...** without an **on** clause) will be invoked before the invocation of the actions for **on button1 double click**.

The **on activate** response statement is invoked when a region become the active region as a result of a user selecting it, or one of its children, with the mouse or the keyboard. If a region is activated due to the execution of an EASEL OS/2 EE statement, the **on activate** response for that region is not invoked. An **on activate** response is *not* invoked when a region is first created.

If there is no **on close** response to invoke for a graphical, image, or textual region, double-clicking on the system menu or selecting the "Close" choice from the system menu destroys the region. If there is no cancel pushbutton response and **on close** response to invoke for a dialog region, double-clicking on the system menu or selecting the "Close" choice from the system menu destroys the region.

The EASEL OS/2 EE compiler will accept any **on** clause syntax for any response to an object or class name. However, note that some object/**on**-clause combinations are not useful, and will never be executed. The following list describes the combinations that are meaningful.

Object	on Clause	Notes
Graphical region, Image region, Textual region	all	A system menu attribute is required to invoke an on close response .
Sense region, Key	on button1 down on button1 double click	
Dialog box	on activate	Pressing the ESCAPE key, double-clicking on the system menu, or selecting "Close" from the system menu invokes a response to the cancel push button, if there is one.
Dialog region	on activate on close	Pressing the ESCAPE key, double-clicking on the system menu, or selecting "Close" from the system menu invokes a response to the cancel push button, if there is one. If no response to the cancel push button is found, EASEL OS/2 EE looks for and invokes an on close response for that region.
List box	on button1 down on button1 double click	Taken when the user selects an entry.
Push button, Radio button, Check box	on button1 down	

6.1.4 Responding to Action Bar Items

You specify the **response to item** ITEM_NAME in a response definition that is to be invoked when the user selects an item in an action bar. You can also specify that the response should be taken only for an item in a specific object or object class. (You can also specify the **response to item** ITEM_CLASS_NAME statement in a response definition that is to be invoked when the user selects any member of an item class.) When the item is selected, EASEL OS/2 EE responds by executing the action statements specified in the response definition. The response will be invoked only if the item is selectable (**enabled**) at the time that it is selected.

6.1.5 Using Response Inquiry Built-in Functions

Response inquiry functions are used to discover the characteristics of the stimulus that initiated a response. For example, if the response was stimulated by selecting an object, the value returned by the **object** function is the name of the object that was selected, and the values returned by the **xcoord** and **ycoord** functions are the position of the select. If the response was stimulated by input from the keyboard or an application, the value returned by the **input** function is the data that was received by EASEL OS/2 EE from the keyboard or the application. These inquiries are shown in the following table.

These functions, like all other built-in functions, can be directly within a response definition, or indirectly, within an action subroutine that is called during the response.

Function	Value Returned
ancestry	<i>Objects:</i> String containing the ancestry of the selected object. <i>Items:</i> String containing the object name that contains the item, with its full ancestry.
input	String received from an application or from the keyboard.
object	String containing the name of the object or item for which a response is taken.
parameter	String containing the selected object's parameter.
xcoord ycoord	<i>Textual Regions:</i> Column and line number of the character selected. <i>Graphical Objects:</i> X and Y coordinates of the point selected, in the coordinate system of the object the response is taken for.

6.1.5.1 The ancestry Function

The **ancestry** function lets you find and use the string associated with the ancestry of the object that stimulated the response or the object and its ancestry containing the item that stimulated the response.

In the example below, after the user selects the object Done, the object's immediate parent is deleted (which causes the object Done to also be deleted).

```
response to Done
  delete ancestry
```

Be careful not to confuse the **ancestry** function with the **ancestry of** function. The **ancestry of** function is an object inquiry, returning the string associated

with the object whose name is specified as the argument to the function. You cannot use the **ancestry of** function with items.

For information on the use of the **ancestry** function with classes of objects, see 6.4.8.2, “Using the ancestry and object Functions Together” on page 6-39.

6.1.5.2 The input Function

The **input** function lets you find and use the input data that stimulated the current response. The data can be received from the keyboard or an application; thus, the **input** function can only be used with the **response to char** or **response to line** response definitions.

When EASEL OS/2 EE receives and recognizes this data and takes the specified response, it returns the data as the value of the **input** function, and performs the actions specified in the response definition. For example:

```
response to char from keyboard
    append input to AccumulatedLine
```

In this example, the user is typing characters on the keyboard. As each character is typed, a response is taken, which appends the character to the AccumulatedLine variable.

If the current response is not stimulated by an input string (using the **response to char** or **response to line** response definitions), EASEL OS/2 EE returns the null string (“”) as the value of the function.

6.1.5.3 The object Function

The **object** function lets you find and use the name of the object or item that stimulated the current response. EASEL OS/2 EE returns a string value representing the name of the object or item that was selected. For example:

```
response to Go
    make object yellow    # Equivalent to make Go yellow
    action SendQuery
```

If the current response is not stimulated by selecting an object (using the **response to OBJECT**, **response to CLASS**, **response to item ITEM**, or **response to item ITEM_CLASS** response definitions), EASEL OS/2 EE returns the null string as the value of the function.

The **object** function is particularly useful in manipulating objects that belong to a class.

6.1.5.4 The **xcoord** and **ycoord** Functions

You can use the **xcoord** and **ycoord** functions to find and use the coordinates of the position of the selection that stimulated the current response. If the object that stimulated the current response is a graphical object, EASEL OS/2 EE returns the following values:

xcoord	The X coordinate of the point selected, in the coordinate system of the selected object.
ycoord	The Y coordinate of the point selected, in the coordinate system of the selected object.

In the following example, a graphical region called PlotArea is displayed. The user can select PlotArea and a tick mark will be displayed where the user selected. The user can create a scatter graph, showing exactly where the tick marks are to be placed. This technique is most useful if the selecting device is the mouse, which makes very precise pointing possible.

```
response to PlotArea
  add to PlotArea
    move to xcoord ycoord
    pattern TickMark
```

If the object that stimulated the current response is a textual region, EASEL OS/2 EE returns values as follows:

The value of **xcoord** is the column number of the point selected.
The value of **ycoord** is the line number of the point selected.

Note that there is not necessarily any character at the point selected. The selection might be to the right of the last character in the line, or below the last occupied line.

If the current response is not stimulated by selecting an object (using the **response to OBJECT** or **response to CLASS** response definition), EASEL OS/2 EE returns values of zero for the **xcoord** and **ycoord** functions.

6.2 Working with Blocks

6.2.1 Overview

A Block is an EASEL OS/2 EE action statement which contains a group of response definitions. Using blocks, you can indicate when responses are appropriate and when they are inappropriate. You can define multiple responses to the same stimulus, wherein under certain circumstances EASEL OS/2 EE would take one response, while under other circumstances EASEL OS/2 EE would take a different response.

A block might be active or inactive. If a block is *active*, the responses defined within it can be triggered by appropriate stimuli. If a block is *inactive*, the responses defined within it are not taken, even if the relevant stimuli occur.

Like other action statements, a block can be specified within a response definition or an EASEL OS/2 EE action subroutine definition. A response definition that is inside a block might itself contain another block as part of its set of action statements (these are called *nested* blocks).

An example of a block is shown below. In this example, as soon as the application receives a "Go \n" message, it sends lines of text to EASEL OS/2 EE, which displays them on the monitor. When the application is finished, it sends the character ">" to EASEL OS/2 EE, which then exits the block.

```
begin                                # Begins block.
# Response to entering block:
  response to start
    send "Go \n" to RemoteA
# Response to receiving character ">" from application:
  response to char col 1 ">" from Remote
  leave block
# Response to receiving a line of text from application:
  response to line from RemoteA
    add to TRegion
    insert input "\n"
end                                  # Ends block.
```

As long as a block is active, its responses can be taken over and over again, until the block is exited. The **leave block** statement allows EASEL OS/2 EE to explicitly exit a block. (There are circumstances under which EASEL OS/2 EE leaves a block without this statement; see 6.2.4.2, "Implicitly Leaving a Block" on page 6-14.)

6.2.2 Defining a Block

You can define and use a block anywhere an action statement can be specified, including within action subroutines, loops, and conditional (if/else) action statements. A block definition uses the following model.

Model

```
begin [ guarded | resumable ] [BLOCK_NAME]

    [RESPONSE_DEFINITION]...

end [BLOCK_NAME]
```

The **begin** statement begins a block. The **end** statement ends a block. The response definitions must lie between the **begin** and **end** of the block. For example:

```
begin Send
    response to char "A" from keyboard
        send "A" to RecorderApp
    response to char "B" from keyboard
        send "B" to RecorderApp
    leave block
end Send
```

If you use an identifier to name a block, as shown in the above example, it must be unique within the EASEL OS/2 EE program. There must be nothing else in the program with that name. If you are using nested blocks (described below), each block should be named.

If you specify an identifier in the **begin** statement, you must specify a name in the **end** statement. It must exactly match the name specified in the **begin** statement. For example:

```
begin Startup
    response to Go
        action Startup
    response to Stop
    leave block
end Startup
```

You cannot specify action statements directly in blocks; all action statements must be specified in response definitions. For example, you can specify:

```
begin
    response to Go
        make Hello visible
        send Startup to application
end
```

but not:

```
begin                # WRONG - missing response definition
    make Hello visible
    send Startup to application
end
```

As shown in the model, you can define a block as **resumable** or **guarded**. These keywords are used to control how a block is exited. (See 6.2.6, “Responding to Stimuli and Returning to a Specified Block” on page 6-17 and 6.2.7, “Disabling Responses Outside a Block” on page 6-18.)

6.2.3 Nested Blocks

Nested blocks are important for controlling program flow. They also come into play when a block is exited without an explicit **leave block** statement. When using nested blocks, name each one for clarity’s sake.

Blocks can be nested within other blocks, creating a program that contains many levels of responses, ranging from the innermost to the outermost. One way of understanding nested blocks is to consider the entire EASEL OS/2 EE program itself as a block. The following is an example of a nested block:

Action subroutine:

```
action DialBlock is
    send "AT\r\n" to DJNS
    begin CheckOK1
        response to line "OK" from DJNS
            leave block
        response to line "ERROR" from DJNS
            action ErrorWait
    end CheckOK1
    change Prompt to
        move to (xsize/2) (ysize/2)
        font "large" text center "Dialing ..."
    send "ATDT" Phonenum "\r\n" to DJNS
    begin CheckConnect                # Outer block
        response to line "Connect" from DJNS
        action Logon
```



```

        leave block
    response to line "Error" from DJNS
        action ErrorWait
    response to Quit
        send "\r\n" to DJNS
        begin CheckOK2                                # Inner block
            response to line "OK" from DJNS
                leave block
            response to line "ERROR" from DJNS
                action ErrorWait
        end CheckOK2
    leave block
end CheckConnect

```

When blocks are nested, control may be transferred from one block to any one of the blocks that are outside that block (at a higher level of nesting). There might be more than one block at any given level, but only one of these can be active at any given time. These situations are discussed in the following section.

6.2.4 Leaving (Exiting) a Block

A block can be left (exited) with the explicit **leave block** statement, or, if the block is not **guarded**, when a relevant response is found outside of the block. Once a block is exited, the responses in that block become inactive and cannot be executed again unless the block is re-entered. Action statements that follow that block will not be executed, since program control is now at a higher response level.

6.2.4.1 Explicitly Leaving a Block

If you specify the **leave block** action statement in a response inside a block, EASEL OS/2 EE will terminate the block when the **leave block** statement is executed. Execution continues at the action statement (if any) immediately following the **end** statement for the block in which the **leave block** statement is specified. For example:

```

action Logon is
    send UserName "\r\n" to DJNS
    begin Calidate
        response to line "Enter Password" from DJNS
            leave block
        response to line "ERROR" from DJNS
            action ErrorWait
    end Calidate
# The following is executed only if
# UserName was a valid entry:
send Password "\r\n" to DJNS
begin CheckPn

```

```

    response to line "Logged On" from DJNS
        leave block
    response to line "ERROR" from DJNS
        action ErrorWait
    end CheckPn
    action SetupMenus
    :

```

A **leave block** statement must be specified inside a response definition that is within the definition of the block itself. You cannot specify it in an action subroutine that is called from inside the block.

You can optionally specify a name in the **leave block** statement. This name specifies the outermost block that you wish to leave. EASEL OS/2 EE terminates the block specified in the **leave block** statement, and all blocks nested within it. If you do not specify a name in the **leave block** statement, EASEL OS/2 EE terminates only the innermost active block.

6.2.4.2 Implicitly Leaving a Block

If a block is not **guarded**, EASEL OS/2 EE may transfer control from one block to an outer block even without encountering an explicit **leave block** statement. This happens when there is a stimulus for which no response is defined in the current block, but for which a response is defined in an outer block. If the relevant response is not in the current block, EASEL OS/2 EE searches each outer block, in succession, (that is, each block at an outer level of nesting) for the relevant response. If one is found, EASEL OS/2 EE transfers control to the block which contains the relevant stimulus, and executes the response. For example:

```

response to Query
    make Submenu blue
    begin MainOpts
        response to SubMenu
            make Screen1 invisible
            make Screen2 visible
            begin SubOpts
                response to MainMenu
                    make Screen2 invisible
                    make Screen1 visible
                    leave block
            end SubOpts
        response to Reset
            make Screen2 invisible
            make Screen1 visible
            make SubMenu blue
            make Reset blue
    end MainOpts

```

When the object Query is selected, the response definitions in the MainOpts block become active. If the user then selects the SubMenu object, the SubOpts block becomes active. At this point, if the Reset or SubMenu objects are selected, EASEL OS/2 EE will leave the SubOpts block and respond to the object from within the MainOpts block.

If in a case of this kind, EASEL OS/2 EE leaves the innermost blocks and all intervening blocks that are not able to respond to the stimulus. These blocks are no longer active, because control is never transferred to a block at a lower level of nesting than the current block. If there are no relevant responses in the current block or in any block at a higher level of nesting, no response is taken.

If two or more blocks exist at any given level of nesting, only one of them can be active at any given time. For example:

```

response to A
  begin Block1
    response to B
  :
    response to C
      begin Block2A      # Block2A and Block2B are
        response to D    # equal in the nesting
                        # hierarchy. Thus, if control
  :
        response to E    # is in Block2A and key G
                        # (which is in Block2B) is
  :
        end Block2A      # selected, control will not
                        # be transferred to Block2B.
    response to F        # But if B is selected (which
      begin Block2B      # is in Block1), control will
        response to G    # transfer to Block1, because
                        # it is at a higher level of
  :
        response to H    # nesting.
  :
    end Block2B
end Block1

```

Remember that EASEL OS/2 EE searches for responses in the program from the top to the bottom from the current block through each outer block, in succession. Thus, if two responses exist for the same object, just the first one encountered will be executed. For example, if the following two responses to Show are the only ones in the program:

```
response to Show
  make Show red
  begin
    response to Show
      make Show blue
      leave block
  end
```

Here, the first time Show is selected, the first **response to Show** encountered is the one which makes Show turn red, and the block is entered. Now, if Show is selected again, the first **response to Show** encountered makes Show turn blue, and the block is left.

6.2.5 Responding to a Block Being Left Implicitly

You specify the **response to termination** response definition within a block if you want an action to occur whenever that block is left implicitly.

In the following example, the user has two objects on the screen (Name and Number). If the user first selects Name and then selects Number (which causes the program to implicitly leave the block in the **response to** Name statement), **response to termination** will clean up the screen by making TypeName invisible.

```
enabled graphical region Name size 50 20
  at position 0 100 border

invisible disabled key TypeName at position 10 10 in Name
:
# Continues...

enabled graphical region Number size 50 20
  at position 100 100 border

response to Name
  make TypeName visible
  begin
    response to char "\n" from keyboard
      leave block
    response to char from keyboard
      add to TypeName
      text input
    response to termination
      make TypeName invisible
```

end

```
response to Number
:
```

The **response to termination** response definition cannot contain a block, nor can it contain a reference to any action subroutine which contains a block. If it does, an unrecoverable runtime error may occur, and no error message will be generated.

6.2.6 Responding to Stimuli and Returning to a Specified Block

If you want EASEL OS/2 EE to leave a specific block in response to any stimulus outside the block and then return to the same block immediately after that response has been executed, specify the keyword **resumable** in the block definition. This feature is particularly useful in a complex program, where various stimuli could cause EASEL OS/2 EE to leave a particular block. Specifying a **resumable block** will always bring EASEL OS/2 EE back to that block, without your having to know which response caused EASEL OS/2 EE to leave it.

When a response outside of a resumable block is triggered, EASEL OS/2 EE will first finish any action statements in the current response which is in the resumable block. Next, it will execute the actions in the response that caused the block to be left, and last, it will return to the block and await a stimulus from within that block.

Resumable blocks cannot contain **response to low memory** or **response to termination** response definitions.

In the example below, if the user selects YesKey and then a response outside of the Menu1 block occurs, EASEL OS/2 EE will first finish the **response to YesKey** action statement, then execute the response that caused the Menu1 block to be left, and finally return to the Menu1 block and await a stimulus.

```
response to Menu1
  begin resumable
    response to start # This response is only taken
                      # when Menu1 is selected; not
                      # when control is resumed.
  :
  response to YesKey
    copy ...
  response to NoKey
    clear ...
    leave block
end
```

Or, in the following example, EASEL OS/2 EE will respond to all lines of data from the application, no matter what other responses are taken in the interim:

```
response to start
  start local Sales "data.exe"

response to GetKey
  begin resumable Always
    response to line "data begin" from Sales
      change DataRegion to input
      copy (Count + 1) to Count
    response to line "data end" from Sales
      stop Sales
  end Always

response to Query
  change CountRegion to Count
  change Prompt to
    "You've received " Count " lines of data.\n"
```

If you want EASEL OS/2 EE to leave any block (other than a guarded block) in response to a specific stimulus, see 6.2.8, “Responding to Specified Stimuli and Returning to a Block” on page 6-19, which describes the **interrupt response** response definition.

6.2.7 Disabling Responses Outside a Block

If you want EASEL OS/2 EE to ignore all stimuli it receives outside of the specified block while the block is being executed, specify the keyword **guarded** in a **begin** BLOCK definition. For example:

```
response to Dial
  send "ATDT\r\n" to DJNS
  begin guarded WaitForOK
    response to line "Connect" from DJNS
      leave block
    response to line "ERROR" from DJNS
      action ErrorWait
  end WaitForOK

response to SendQuery
  :
response to line from DJNS
  :
response to Stop
  :
```

When the object Dial is selected, the string “ATDT\r\n” is sent to the application DJNS, and EASEL OS/2 EE will wait for a response to occur in the

WaitForOK block. If SendQuery, Stop, or a line other than “Connect” or “ERROR” occurs, the responses will *not* be taken until the WaitForOK block has been exited.

When a guarded block is being executed, EASEL OS/2 EE holds keyboard and application input which is not applicable to the responses within the block, making them available when the guarded block exits. Selections will not be taken for objects which do not have responses within that block, no matter what level of nesting they occupy.

A guarded block can be left only by the explicit statements **leave block**, **leave loop** (if the guarded block is within a loop), **exit** or **change to program** (which must be specified within the block), or **response to low memory**.

A guarded block cannot contain a **response to low memory** or a **response to termination** response definition. Specifying a **response to termination** in a guarded block does not result in an error message, but will have no effect.

6.2.8 Responding to Specified Stimuli and Returning to a Block

If you want EASEL OS/2 EE to leave any block (except a guarded block) in response to a specified stimulus and then return control to that block immediately after the response has been executed, specify the keyword **interrupt** in the response definition. Note that specifying an **interrupt response** within a block itself serves no purpose, unless the block contains other (nested) blocks.

When an **interrupt response** occurs, EASEL OS/2 EE will first finish any action statements in the current response, then execute the actions in the interrupt response, and finally return to the block and await a stimulus from within that block. An interrupt response affects all blocks *except* guarded blocks. You cannot use an **interrupt response** statement for the **response to start**, **response to low memory**, or **response to termination** response definitions.

In the example below, if the user selects Menu1, then YesKey and then selects Menu3, the Menu3 response will first wait until the **response to YesKey** response statement has finished, then execute the response to Menu3, and last, return to the Menu1 block and await a stimulus.

```
response to Menu1
  begin
    response to YesKey
  :
```

```

        response to NoKey
    :
        leave block
    end

interrupt response to Menu3
    make object blue

response to Menu4 # if user selects YesKey and
                  # then Menu4, control will not
                  # be returned to the Menu1 block.

```

6.2.9 Using the response to start Response Definition Within a Block

Just as **response to start** is used in the EASEL OS/2 EE program to perform certain operations at program startup, **response to start**, when defined with a block, is used to perform specified operations immediately after the block becomes active. For example:

```

response to File
    begin FileBlock
        response to start
            make File yellow
            make FilePull visible
            change Prompt to
                font "large" text center
                "Select the file name"
        response to FileText
    :
        response to FileOK
    :
        leave block
        response to FileQuit
    :
        leave block
    end FileBlock

```

Although the use of the **response to start** definition is recommended, the action statements or action subroutines in the **response to start** can be placed before the block is entered, with the same results. For example:

```

response to File
    make File yellow
    make FilePull visible
    change Prompt to
    :
    read file ...
    begin FileBlock
        response to FileText
    :

```



```

·
  response to FileOK
:
  response to FileQuit
:
end FileBlock

```

Just as there can be only one **response to start** in the EASEL OS/2 EE program to respond to program startup, there can be only one **response to start** within each block.

6.2.10 Using the response to char Response Definition in Blocks

Although the **response to char** response definition is not described until 14.2.2.2, “Responding to Character Input” on page 14-9, there are special considerations when using blocks that contain this response which should be noted here. In this situation, you must be careful to ensure that EASEL OS/2 EE knows when no more information is expected, so that all available responses—including those outside the block—can be taken. In order to check for a pattern match in an outer block, the pattern in the inner block must not be matched. When you are using **response to char**, the *only* way to do this is by using a specific pattern match that excludes any other input, i.e., using a column specification as shown below.

In the following example, if “y” is not typed, the block will never be left, and no other responses from characters typed from the keyboard can be taken.

```

response to ExitPrompt
  begin
    response to char "y" from keyboard # inner
      send "y\n" to Application1      # block
      leave block
    end
  end

interrupt response to char "n" from keyboard # outer
  send "n\n" to Application1           # block

```

In order to make sure that all available responses can be taken, use the **col** specification in the match clause (described in detail in Chapter 14). In the following example, if the user types any character *other* than “y” in column 1, the block will be left and EASEL OS/2 EE is free to take the response to the user’s typing “n.”

```

response to ExitPrompt
  begin
    response to char col 1 "y" from keyboard
      send "y\n" to Application1
      leave block
    end
  end

response to char "n" from keyboard

```

send "n\n" to Application1

response to char from keyboard
append input to StringA

6.3 About Action Statements

6.3.1 Overview of Actions and Their Uses

In many ways, action statements are the heart of an EASEL OS/2 EE program. The instant that a response statement is activated, it in turn activates a set of action statements which control the processing, timing, and interaction among all parts of the EASEL OS/2 EE interface. Some action statements perform operations on objects—for example, changing attributes or contents of objects, adding or deleting objects, or changing class memberships. Other action statements perform such operations as assigning values to variables; calling subroutines; starting, stopping, and communicating with application programs; performing calculations; aiding in program debugging; and ending the EASEL OS/2 EE program.

6.3.2 Specifying Action Statements

All action statements can be executed during runtime only, and must be contained within a response definition or action subroutine. In most action statements, you supply the name (identifier) of the object upon which the statement is to act. For example, the statement:

```
response to Reduce
  delete Triangle
```

deletes the definition of the object named Triangle (and, in this case, all of Triangle's children). You can specify either a single object or a class of objects; if you specify a class, the statement acts on all objects in the class (see 6.4.4, "Specifying Actions for a Class of Objects" on page 6-36).

In place of a name, you can specify a string variable (including any of the valid built-in functions) that is to be converted to a name. EASEL OS/2 EE performs this string-to-name conversion. For example, you can specify any of the following:

```
response to line from keyboard  # The typed lines are
  add to TextDisplay           # displayed in the
  insert input                 # region TextDisplay.
```

or:

```
response to DisabledKeyClass
  delete object                # object is a built-in function.
```

or:

```

response to ...
  copy DisabledKey to DisabledKeyName
  append Counter to DisabledKeyName
                                # DisabledKeyName is a string
  delete DisabledKeyName        # variable.

```

6.3.3 Creating, Changing, Clearing, and Deleting Objects

This section presents the basic action statements that you can specify for any object. They create a new object or delete an existing one, change the contents of objects, and change the attributes of objects. (Other action statements associated with particular features—for example, textual regions, graphical objects, classes of objects, communications, etc.—are described in each of the appropriate chapters.) These are:

add	Adds a new object.
add to	Adds new drawing statement(s) without clearing existing ones.
change	Adds new drawing statements, clears existing ones, and deletes all of the object's children.
change graphics	Adds new drawing statements and clears existing ones, without deleting its children.
clear	Deletes contents of object and its children.
clear graphics	Deletes contents of object, without deleting its children.
delete	Deletes object and its children.

6.3.3.1 Creating a New Object

The **add** action statement allows you to create a new object. In the **add** statement, you define an object exactly as you would in a regular object definition. The new object is created when the **add** statement is executed. (Be careful to not confuse the **add** action statement with the **add to** addition statement, which adds contents to an existing object.)

The **add** statement is commonly used when you do not know how many objects you will be creating. For example, you might want to create an object for each line of input that EASEL OS/2 EE receives from an application, as shown in the example below:

```

response to line from application
  extract from input
    take word Name
    take number X
    take number Y
  add key Name at position X Y
  pattern KeyPattern

```

EASEL OS/2 EE extracts fields from an input line received from an application and stores them in the variables called Name, X, and Y. For example, if the application sends the following text:

```
A3 216 516
```

EASEL OS/2 EE extracts a word, A3, which becomes the value of the variable Name. The numbers 216 and 516 are extracted and stored as the values of the X and Y variables. The **add** statement creates a new key whose name is A3 and whose position is 216 516. Each time EASEL OS/2 EE matches the input to the specified control string, new values are extracted for the Name, X, and Y variables, and a new object is created.

When adding an object during runtime, be careful to ensure that the program does not create multiple objects with the same name. This can happen when a response is executed multiple times. To minimize this risk, specify a variable, rather than an explicit object name, in the **add** statement.

You can add an object to a class of objects with an **add to class** statement.

6.3.3.2 Adding to the Contents of an Object

You can add to the contents of an object (or to the contents of all of the objects in a class) by specifying an **add to** statement. The contents of an object are described by the drawing statements specified in the object definition.

When you specify an **add to** statement, EASEL OS/2 EE preserves all of the existing drawing statements for the object, and adds the new statements. For example:

```
response to EnterPassword
  clear PasswordRegion
  begin
    response to char from keyboard
      append input to Password
      add to PasswordRegion
        pattern Asterisk
      response to PasswordDone
        leave block
    end
```

For graphical objects, the **at position** specification in an **add to** statement is equivalent to specifying a **move to** drawing statement as the first item in the contents to be added. If the **at position** specification is omitted, the new drawing statements begin at the current graphics cursor or text cursor position. For example:

```
response to SelectArea
  add to SelectArea at position xcoord ycoord
    pattern CrossHatch
```

This is equivalent to:

```
response to SelectArea
  add to SelectArea
    move to xcoord ycoord
    pattern CrossHatch
```

Since a textual region's drawing statements are distinct from graphical drawing statements, do not specify an **add to** statement for a class of objects that includes textual regions as well as graphical objects.

6.3.3.3 Deleting the Contents and Children of an Object

You can delete the contents and children of an object, or all of the objects in a class, by specifying a **clear** statement. The position of the cursor(s) are restored to the object's origin. For example:

```
response to Restart
  clear AddKey
```

Before you clear an object, it is recommended that you first delete the object and its children from any classes to which they belong, using the **delete from class** statement.

6.3.3.4 Deleting the Contents of an Object Without Deleting Its Children

The **clear graphics** statement is the same as the **clear** statement, except that the **clear graphics** statement does not delete, or affect in any way, any of the object's children. Only the contents of the specified object are deleted.

6.3.3.5 Replacing the Contents of an Object and Deleting Its Children

You can replace the contents (drawing statements) of an object, or the contents of all of the objects in a class, by specifying a **change** statement. EASEL OS/2 EE clears the existing drawing statements for the object, deletes all of the object's children, and adds the new drawing statements starting at the object's origin. This statement has the same effect as using the **clear** statement followed by the **add to** statement. For example:

```
response to Logoff
  change Message to
    text center "Logging off, please wait"
```

Since a textual region's drawing statements are distinct from graphical drawing statements, do not specify a **change** statement for a class of objects that includes textual regions as well as graphical objects.

Before you replace the contents of an object, it is recommended that you first delete the object and its children from any classes to which they belong, using the **delete from class** statement.

6.3.3.6 Replacing the Contents of an Object Without Deleting Its Children

The **change graphics** statement is the same as the **change** statement, except that the **change graphics** statement does *not* delete, or affect in any way, any of the object's children. For example, if there is a region named PanelArea that displays some report options and you want to modify PanelArea but not delete its children (the report options), you should use the **change graphics** statement:

```
response to PanelArea
  change graphics PanelArea to # Reflects active mode
    move 10 10
    solid blue box 100 80
    move 10 10
    solid color 20 box 80 60
  :
response to Reset
  change graphics PanelArea to # Reflects inactive mode
    move 10 10
    solid green box 100 80
  :
```

6.3.3.7 Deleting an Object

You can delete an object (or all the objects in a class) by specifying **delete** statement. Deleting an object deletes the object itself, its contents, and its children. For example:

```
delete Water in Liquids in Coolants
```

Here, Water is deleted, along with any children Water may have. Liquids and Coolants are not affected by the statement (except that Water is no longer a child of Liquids, because it no longer exists).

Before you delete an object, you must first delete the object (as well as its children) from any classes of which it is a member, using the **delete from class** statement. If you do not, performing an action on the class will produce an error during runtime.

If you wish to delete the contents and children of an object, but want to retain the object itself with all of its other attributes, use the **clear** statement

6.3.4 Changing Attributes of Objects

6.3.4.1 Changing Selectability

You can change the selectability of an object, or of all of the objects in a class, by specifying an **enable** or **disable** statement. An enabled object is selectable; a disabled object is not. For example:

```
response to Delete  
  enable DeleteQuit
```

You can use the **selectability of** built-in function in an action statement to find out an object's current state of selectability during runtime.

6.3.4.2 Changing Visibility

You can change the visibility of an object (or the visibility of all of the objects in a class) by specifying a **make visible** or **make invisible** statement. For example:

```
response to Go  
  make StopSign visible
```

You can use the **visibility of** built-in function in an action statement to find out whether an object is currently visible or invisible.

6.3.4.3 Changing Color

You can change the color of a key or the background color of a graphical or textual region, by specifying a **make** COLOR statement. You can also specify a **make** COLOR statement to change the color of all of the objects in a class. For example:

```
response to Logo
  make Screen1 blue
```

Here, Screen1 is changed to blue.

You can use a string variable for the color; for example:

```
make Lobster in Seafood color ColorChoice
```

Here, ColorChoice is a variable which may contain a string value representing one of the color keywords, or an integer value representing one of the color numbers.

This statement changes the default color of the object's drawing statements, but does not override the explicit colors specified within the drawing statements. For example:

```
blue key BigKey at position 100 200
  solid box 50 50
  move to 10 20
  black text align "BIG"
```

```
response to MakeRed
  make BigKey red
```

This changes the color of the solid box, but does not change the color of the text. If you want to change the color of the object's drawing statements (such as the text in BigKey), you need to specify a **change** or **change graphics** action statement.

You can change the color of an object (or all the objects in a class) by adding a color value to the **make** statement. For example:

```
response to BigPicture
  make object blue
```

You can include ancestry in the statement, and you can use a string or integer variable to hold the color value; for example:

```
make Graph in BigPicture color ColorChoice
```

Here, Graph becomes and its contents are drawn in the color specified for the variable ColorChoice.

You can change the color of a **colored textual region**. For example:

```
make Speedometer in Dashboard foreground blue
```

To change both the background and foreground color, you need to write separate statements; for example:

```
make Notebook foreground aqua
make Notebook background red
```

Note that when you change the foreground or background of a textual region, *all* previous color specifications—including those used to color individual characters in a colored textual region—are cleared and replaced with the newly specified color.

You can use the following built-in functions to find out the current color of an object during runtime.

foreground of	Returns the foreground color of the specified object.
background of	Returns the background color of the specified graphical or textual region.
border of	Returns the color of the border of a specified graphical or textual region.
foreground at	Returns the foreground color of a specified position in a textual region.
background at	Returns the background color of a specified position in a textual region.

6.3.4.4 Changing Border Attributes in Regions

You can change the color of the border of a graphical or textual region (or of all the borders in a class) by specifying a **make border** COLOR statement; for example:

```
make Inventory border color ColorChoice
```

You can change the visibility of the border by specifying a **make border visible** or **make border invisible** statement; for example:

```
make Tennis in Sports border invisible
```

To change the border's visibility and color, you must write two statements; for example:

```
response to ...
  make Inventory border visible
  make Inventory border red
```

You can use the **border of** built-in function to find out the current color of a border during runtime. If the value returned is negative, the border is also invisible.

6.3.4.5 Changing Ancestry

An object's ancestry can be changed with the **change position** statement.

You can use the **ancestry of** built-in function to find out an object's current "path" of ancestors during runtime.

6.3.4.6 Changing Priority

You can change the priority of an object (or of all of the objects in a class) by specifying the **change priority** statement. You specify the priority as an integer value. For example:

change Button1 priority to 3

You can use the **priority of** built-in function to find out an object's current priority level (number) during runtime.

6.3.4.7 Changing Parameter

You can change the parameter associated with an object (or the parameter associated with each of the objects in a class) by specifying the **make parameter** statement. You specify the new parameter as a string value. After EASEL OS/2 EE has executed the **make parameter** statement, the value that is returned when you use the built-in **parameter** function is the new parameter of the object. For example:

make BlueKey parameter "ColoredKey"

6.3.5 Changing the Size and Position of an Object

6.3.5.1 Changing Size

Changing the size of a viewport or window of a graphical region is described in 7.3, "Action Statements for Changing Windows and Viewports in Graphical Regions" on page 7-47; changing the size of a viewport or window of a textual region is described in 8.4.9, "Changing the Size and Position of a Textual Region's Window and Viewport" on page 8-36.

6.3.5.2 Changing Position

You can change the position of a key or the position of the viewport of a graphical or textual region, by specifying a **change position by** statement or a **change position to** statement. You can also specify these statements to change the position of all of the objects in a class.

```
change { OBJECT_NAME | CLASS_NAME } position  
    { by XOFFSET YOFFSET | to X Y [in PARENT_NAME] }
```

Remember that the position of each of these objects is specified in the coordinate system of its parent (which may be the screen itself).

The **change position by** statement describes the offsets from the object's current position to its new position. For example, if you have defined the following objects:

```
graphical region Figure size 400 300 at position 0 0
```

```
key Square at position 100 100 in Figure  
    box 50 50
```

and then change Square's position as follows:

```
change Square position by 50 100
```

Square's new position will be coordinate position 150 200 in Figure.

The **change position to** statement describes the absolute position of the object, in its current parent or in a new parent, regardless of the object's current position. In the following example, Square is moved to position 110 120 in Figure.

```
change Square position to 110 120
```

If you have defined the following objects:

```
graphical region Menu1 size 100 100 at position 20 40
```

```
key Button1 at position 70 20 in Menu1  
    text "Button #1"
```

and then specify:

```
change Button1 in Menu1 position by -50 0
```

Button1, a key in Menu1, is located at position 70 20 within Menu1's viewport. When you change the location of Button1 by -50 0, Button1 is moved to the left by 50 positions and remains stationary in the Y direction. After the **change position by** statement has been executed, Button1 will be located at position 20 20 within Menu1's viewport.

6.3.5.3 Changing Ancestry in the change position to Statement

You can also specify new ancestry using the **change position to** statement. For example, if you have defined the following object:

key Zipper at position 10 10 in OldCoat

and then specify:

change Zipper position to 100 300 in NewCoat

New ancestry is specified for Zipper. This statement removes Zipper as the child of OldCoat and adds Zipper as the child of NewCoat. Zipper's origin is now position X=100 Y=300 in the coordinate space of NewCoat.

6.4 Using Classes of Objects

6.4.1 Overview

A **class** is a group of objects. Classes are used for:

- Performing one action for many objects, using just one action statement.
- Responding to many objects, using just one response statement.
- Dynamically changing the response definition for an object. During runtime, you can change the composition of a class by adding objects to the class and deleting objects from it. By moving an object from one class to another, you effectively change the response that is to be taken for that object.

As a general rule, it is inadvisable to group different types of objects as members of the same class. You should define classes that consist of only one type of object; for example, classes that consist of only keys or only graphical regions. This is especially true for textual regions, which have many unique action and drawing statements, and cannot use most of the action and drawing statements that are used by graphical objects.

6.4.2 Defining a Class of Objects

You define a class, and provide the names of the objects that are members of the class, by specifying a **class** definition. All of the objects specified in the **class** definition must have been previously defined in the EASEL OS/2 EE program. For example:

```
key Print at position 250 200
:
key Graph at position 250 150
:
key UpArrow at position 250 250
:
key DownArrow at position 250 100
:
key Help at position 250 50
:
class MenuKeys is
    Print Graph UpArrow DownArrow Help
```

A class can be initially null. The set of objects defined as being members of a class is not static; during runtime, you can add objects to, and delete objects from, a class.

An object can be a member of more than one class. For example:

```
class Numbers is One Two Three Four Five Six Seven Eight
class Choices
```

```
response to Numbers
  add object to class Choices
  make object blue
```

```
response to Choices
  send parameter to Focus
  make object yellow
```

You cannot define a class as a member of another class; only objects can be class members. You can add a class of objects to another class, but this simply adds all the objects of the first class to the second class.

6.4.3 Defining Responses for a Class of Objects

You can define a response for all of the objects that make up a class. In the following example, a confirmation message will be displayed if the user selects any one of the keys that are members of the MenuKeys class:

```
response to MenuKeys
  make KeyMessage visible
```

In general, you want to group, into a class, objects for which the same response will be taken. Note, however, that you can specify individual responses for the individual objects in a class. In the following example, Help is a part of the class MenuKeys, but when Help is selected, only that individual response will be taken. Other members in the class MenuKeys will not be affected by the response.

```
response to Help
  make PanicMessage visible
  exit
```

Remember that EASEL OS/2 EE searches for responses in the program from the top to the bottom from the current block through each outer block, in succession, and, if two responses exist for the same object, just the first one encountered will be executed. Thus, when an object is selected, the first relevant response will be executed. For example:

```
response to MenuKeys # Help is a member of the
                    # class MenuKeys ...
:
response to Arrows   # ... and is a member of the class
                    # Arrows ...
:
response to Help     # ... and has a response of its own.
:
```

Because of the order in which these statements are written, only the **response to MenuKeys** response statement will be executed when the Help key is selected. You can control the order in which responses are taken by using blocks.

Often, you will want to specify a single response for an entire class of objects, but to vary slightly the actions that are performed for the objects that make up the class. One way to do this is to use the built-in functions **parameter** and **object** in the response definition for the class. See 6.4.8.4, “Using the parameter and object Functions Together” on page 6-40.

6.4.4 Specifying Actions for a Class of Objects

In general, any action statement that you specify for an individual object can also be specified for a class of objects. The following provides examples of representative statements:

```
delete Positions    # Deletes all of the objects in the
                   # Positions class.

clear Positions     # Deletes the contents of the objects
                   # of the Positions class, but does not
                   # delete the objects themselves.

add to Keys         # Adds a move statement to the object
  move to 0 0       # definitions of all of the objects
                   # in the Keys class.

make DangerGauges red  # Changes to red the colors of
                      # all of the objects in the
                      # DangerGauges class.

make Messages invisible # Makes all of the objects in the
                      # Messages class invisible.

enable Buttons      # Makes all of the objects in the
                   # Buttons class selectable.
```

CAUTION:

Be careful in using action statements for objects that belong to classes, or for objects whose children belong to classes. Some action statements delete children: for example, before you use the change, clear, or delete action statements for an object, you must first delete its children from any classes to which they belong, using the delete from class action statement.

In addition, there are two action statements unique to classes: the **add to class** and the **delete from class** statements.

6.4.5 Adding an Object to a Class

Once you have defined a class, the **add to class** action statement lets you add any object to the class. The object must already exist; it may have been previously defined, or it may have been created during runtime using the **add** statement. (Note that the **add** statement, which defines an object during runtime, is different from the **add to class** statement described here, which adds an existing object to a class.) For example:

```
add Small to class Size
```

Ancestry can be specified in the **add to class** statement; for example, the following code adds the object Monkey, which is a child of Animals, to the class JungleBeasts:

```
add Monkey in Animals to class JungleBeasts
```

An object can be a member of many classes simultaneously. Adding an object to one class has no effect on the object's membership in other classes.

If a response has been defined for the class, then an object that is defined and added to the class at runtime can stimulate the response when it is selected. For example:

```
string variable KeyName
integer variable Counter is 0
:
response to ...
  copy "BlueKey" to KeyName
  append Counter to KeyName
  add key KeyName at position 50 50
  add KeyName to class BlueClass
  copy (Counter + 1) to Counter
:
```

KeyName is a variable to which the string literal "BlueKey" was assigned. The first time this response is performed, the **add** statement creates a new object whose name is BlueKey0. The second time the response is performed, BlueKey1 is created, and so on.

An object can be created during runtime, and added dynamically to a class. In the following example, an object is created during runtime, at the coordinate positions entered from the keyboard.

```
# Define an initially empty class:
class Keys
:
response to line from keyboard
  # Extract X and Y positions from keyboard input:
  extract from input
  take number X
```

```

        take number Y
    copy (Counter+1) to Counter
    # Construct name of key:
    copy "Point" to Name
    append Counter to Name
# Add newly created key at specified position:
    add key Name at position X Y
    :
# Add constructed key name as member of Keys class:

add Name to class Keys

```

When you specify an object name in the **add to class** action statement, make sure that the name does not already exist within that class; if it does, you must first remove the object from the class and then re-add it. Otherwise, the addition has no effect and only the old object (with the same name) will remain in the class.

6.4.6 Deleting an Object from a Class of Objects

You can delete an object, or even a class of objects, from a class of objects by specifying a **delete from class** statement. For example:

```
delete Rolls from class LuxuryCars
```

Here, the object Rolls is deleted from the class LuxuryCars. Or:

```
delete Fuels from class HeatingCosts
```

Here, each of the members of the class of objects, Fuels, is deleted from the class HeatingCosts.

Deleting an object from a class has no effect on the attributes of the object itself. The object still exists. The only results of deleting an object from a class is that selecting the specified object no longer invokes the responses that have been defined for the class, and any action statements that reference the class no longer affect that object.

When an object is a member of more than one class, deleting it from one class has no effect on its membership in other classes.

6.4.7 Changing Class Membership

Moving an object from one class to another provides a means of changing, during runtime, the responses that are performed for a particular object. For example:

```

class OccupiedBeds is Bed1 Bed2 Bed3

class VacantBeds is Bed4 Bed5 Bed6 Bed7

response to MoveOut1
  delete Bed1 from class OccupiedBeds
  add Bed1 to class VacantBeds

```

If MoveOut1 is selected, Bed1 is deleted from the OccupiedBeds class and added to the VacantBeds class. After this response has been taken, selecting Bed1 no longer invokes the response defined for OccupiedBeds. Instead, it invokes the response defined for VacantBeds. (Note, however, that any response to Bed1 is still invoked when the key is selected.)

6.4.8 Using Built-in Functions with Classes

6.4.8.1 Using the object Function with a Class

You can tell EASEL OS/2 EE to perform the specified action statements when the user selects any object that is a member of the specified class, by specifying the **object** built-in function. For example, suppose you have defined a class named Buttons:

```
class Buttons is Button1 Button2 Button3 Button4
```

If the following response definition is specified:

```

response to Buttons
  make Buttons magenta
  make object white

```

then selecting one of the selectable buttons in the class—Button1, Button2, Button3, or Button4—results in all the buttons selected turning magenta; then the one selected turning white.

6.4.8.2 Using the ancestry and object Functions Together

If the **object** function is specified more than once within a single response, EASEL OS/2 EE returns the same value until the response is complete. For example, if your program contains two objects such as the following:

```

key Help at position 100 0 in Screen1
  solid box 100 80
  move 50 40
  text center "Help"

```

```

key Help at position 100 0 in Screen2
  solid box 100 80
  move 50 40
  text center "Help"

```

you should use the **ancestry** function to determine which object was selected:

```
response to HelpKeys
  copy object in ancestry to ObjectSelected
  make ObjectSelected invisible
:
```

The **object** function does not return the ancestry along with the name. In the case where you are using the **object** function for two objects with the same name, use the **ancestry** function as well; for example:

```
copy object in ancestry to CurrentObject
```

6.4.8.3 The parameter Function

Object parameters are very useful when you wish to specify a response definition for a class of objects, and then perform distinct action statements for each of the members of the class. In such a case you can specify, as the **parameter** attribute, an identifying string. In the following example, the parameter of each object is used to have different cities become visible.

```
key Northeast ...
  parameter is "Boston"

key Midwest ...
  parameter is "Chicago"

key Southeast ...
  parameter is "Atlanta"
```

```
class Districts in Northeast Midwest Southeast
```

```
response to Districts                # When Districts is
  copy parameter to Headquarters      # selected, the city
  make Headquarters visible           # name is displayed.
```

Be careful not to confuse the **parameter** function with the **parameter of** function. The **parameter of** function is an object inquiry, returning the parameter string associated with the object whose name is specified as the argument to the function.

6.4.8.4 Using the parameter and object Functions Together

Often, you will want to specify a single response for an entire class of objects, but to vary slightly the actions that are performed for the objects that make up the class. The built-in functions **parameter** and **object** are especially useful in this situation, to customize the actions performed for the objects in the class. For example:

```

graphical region InputArea ...

graphical region Keypad ...

key One at position ... in Keypad
:
  parameter is "1"

key Two at position ... in Keypad
:
  parameter is "2"

class NumberKeys is One Two ...

response to NumberKeys
  if (object = "EnterKey") then
    make InputArea invisible
  else
    add to InputArea
    text parameter
  end if

```

When any one of the keys that are members of the `NumberKeys` class is selected, the **response to NumberKeys** response statement is performed. But the specific actions depend upon the particular key that is selected. The value returned by the **parameter** function is different for each key. If the One key is selected, for example, the number "1" is displayed in the region InputArea. (The **if/else** conditional action statement is discussed in Chapter 11.)

6.4.8.5 Using the members of Function with a Class

The **members of** function can be applied to any class name. The value returned by EASEL OS/2 EE is a string of names of all the objects in the specified class, separated by spaces. For example, if you have defined a class named `PrintColors` and you wanted to reposition these objects, you would use the **members of** function:

```

string PrintStr
integer XPos

class PrintColors is PrintRed PrintGreen PrintBlue
:
response to Change
  copy 100 to XPos
  extract from members of PrintColors
    take word PrintStr # Reset marker; get first object.
  change PrintStr position to XPos 40 in Print2
  copy (XPos + 100) to XPos

```

```

while (marker < length of members of PrintColors) loop
  extract from members of PrintColors
  skip to marker
  take word PrintStr
  change PrintStr position to XPos 40 in Print2
  copy (XPos + 100) to XPos
end loop

```

If any name in the class is ambiguous, EASEL OS/2 EE will return the object's entire ancestry for clarification. For example, if you have defined two objects and a class as follows:

key One at position 10 10 in Calculator

```

Class Keys is
  KeyA KeyB One

```

key One at position 10 10 in DigitPad

and then later specify in your program the function:

members of Keys

EASEL OS/2 EE returns the string "KeyA KeyB Calculator/One".

If you specify ancestry when you add an object to a class, the **members of** function will return the object name along with the ancestry you specified, even if the object name is not ambiguous.

Chapter 7. Graphical Objects, Image Regions, and Sense Regions

7.1 Graphical Objects

There are eight kinds of EASEL OS/2 EE objects: keys, graphical regions, image regions, textual regions, sense regions, dialog regions, dialog boxes, and dialog controls. Of these, keys and graphical regions are graphical objects.

7.1.1 Defining Keys

A key is most often used to trigger a response when it is selected; therefore, it is enabled by default. A disabled key is usually displayed to provide the user with information.

The initial attributes of a key are specified in a **key** definition. You can change any of these attributes during runtime. The definition uses the following model.

Model

```
[ enabled | disabled ] [ visible | invisible ] [ [color] COLOR]
key KEY_NAME
  at { [position] X Y | default position }
  [in PARENT_NAME1 [in PARENT_NAME2]... ]
  [priority is PRIORITY_NO]
  [parameter is PAR_STRING]

  [DRAWING_STATEMENT]...
```

When specifying attributes, the **priority** and **parameter** can be specified in any order. Other attributes must be specified in the order shown in the model. Only the **key** specification, the identifier, and the **at [position]** specification are required. Other attributes are optional, using the defaults shown below.

```
SELECTABILITY:  enabled
VISIBILITY:    visible
COLOR:         white
ANCESTRY:      in primary region (in master for fullscreen)
PARAMETER:     ""
PRIORITY:      0
```

Thus, if you define a key with just the required specifications, such as:

```
key Stop at 25 10
```


EASEL OS/2 EE compiles this key definition as:

```
enabled visible white key Stop
  at position 25 10
  priority is 0 parameter is ""
```

Although EASEL OS/2 EE provides defaults for these attributes, you may find it useful to specify attributes that the object would be given by default; it reminds you, and anyone working on your program, of all the attributes of each object.

The drawing statements that you can use to define the initial appearance of a key are presented in 7.2, "Drawing Statements for Graphical Objects" on page 7-9.

7.1.1.1 Creating New Identifiers for Objects

An object can be defined with an identifier (a capitalized name), or a string value which will be evaluated when the object is created. For example, if you want to create a new object every time a particular response is taken, you can specify:

```
string variable KeyName
integer Counter is 1
:
# Continues...

response to CreateBox
  copy "Box" to KeyName
  append Counter to KeyName
  add key KeyName at position 200 200
  copy (Counter + 1) to Counter
```

7.1.1.2 Specifying Position in the Object Definition

The position defines the origin of the object in its parent. You specify the position of a graphical object by specifying the **at position** attribute in the object's definition, or by specifying **at default position**.

When you specify the **at position** attribute, you define the object's origin in the X and Y coordinates in the coordinate system of the object's parent. If the object has no parent, the default parent is the primary region in windowed applications; master in fullscreen applications. For example:

```
key Start at position 50 100
```

You can also add ancestry to the position, placing the object in the coordinate system of another object; for example:

```
key Start at position 50 100 in OpenFinance
```

Here, the coordinate positions of the child `Start` are relative to the starting position of its parent, `OpenFinance`.

You can specify an integer value to represent the coordinate positions; for example:

```
key Start at position XStart (Height*2)
```

When you specify the **at default position** attribute, you define the object's origin as being at the current position of the graphics cursor of the object's parent. If the object has no parent, the default position is 0 0 ($X=0$, $Y=0$) in the primary region or master.

The **at default position** specification is useful when you add or modify an object in a response definition, using the **add** or **change to** statements. Using this specification, you need not compute the exact position of the graphics cursor in the parent object. For example, if you add an object as the child of a graphical region, you can specify the child's position as **at default position**, as shown below:

```
graphical region Box1 size 50 50 at position 0 0
  draw 10 10
```

```
response to start
  add key Box2 at default position in Box1
    box 20 20
```

In this example, `Box2` is at position 10 10 in `Box1`, because that is where the last drawing statement in `Box1` left its graphics cursor.

7.1.1.3 Color in Keys

When you specify a color for a key, you are defining the default color of the key's drawing statements. Thus, any drawing statements that do not have a specified color will be drawn in the color defined in the key definition.

7.1.2 Defining Graphical Regions

A graphical region is a rectangular portion of the screen in which other objects can be drawn. A graphical region can have any of the attributes that you specify for keys (selectability, visibility, priority, parameter, and color). In addition, a graphical region has special attributes not applicable to keys; these are **size**, **background** color, and **border**.

Because a graphical region can have both a background color and a border color, you can create a logical division on the screen by defining one or more graphical regions and displaying other objects in them. In its simplest usage, a graphical region is a way of painting all or part of the screen, and of setting off the painted area by displaying a border of a different color.

The initial attributes of a graphical region are defined in a **[graphical] region** statement. You can change any of these attributes during runtime. The definition uses the following model.

Model

```
[ enabled | disabled ] [ visible | invisible ] [ [color] COLOR]
[primary] [graphical] region GR_NAME size XSIZE YSIZE
  at { [position] X Y | default position }
  [in PARENT_NAME1 [in PARENT_NAME2]... ]
  [window size WINDOW_XSIZE WINDOW_YSIZE
    at [position] WINDOW_X WINDOW_Y]
  [ [color] COLOR foreground]
  [priority is PRIORITY_NO]
  [parameter is PAR_STRING]

[REGION_ATTRIBUTES]

[DRAWING_STATEMENT]...
```

The REGION_ATTRIBUTES that you can use to define the appearance of a graphical region are presented in 10.1, "Region Attributes" on page 10-2.

Any of the attributes after ancestry can be specified in any order.

Only the **region** specification, the identifier, the **at [position]** specification, and the **size** specifications are required. Other attributes are optional, using the defaults shown below.

SELECTABILITY:	enabled
VISIBILITY:	visible
COLOR (background):	black
ANCESTRY:	in primary region (in master for fullscreen)
COLOR (foreground):	white
PRIORITY:	0
PARAMETER:	""
BORDER:	invisible
COLOR (border):	white

Thus, if you were to define a region with just the required specifications, such as:

```
region Store size 50 50 at position 100 100
```

In a windowed application, EASEL OS/2 EE would read this region as:

```
enabled visible black graphical region Store
  at position 100 100
  size 50 50
  white foreground
  priority is 0 parameter is ""
```

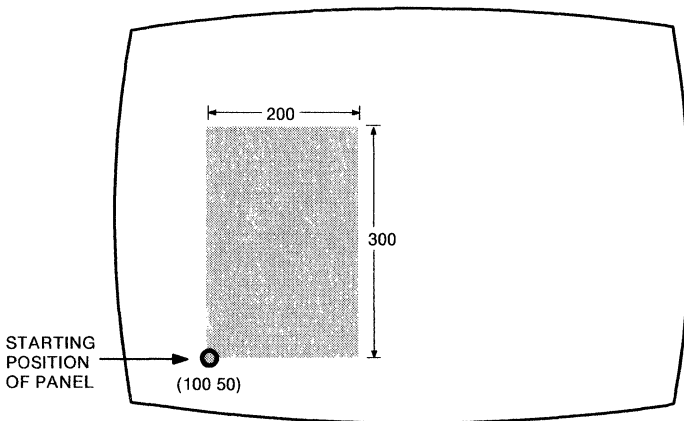
The parent defaults to the primary region.

7.1.2.1 Defining the Size and Position of a Graphical Region

When you define a graphical region, you indicate its size by specifying its X and Y dimensions in the coordinate system of its parent. For example, the following graphical region definition tells EASEL OS/2 EE that the graphical region is a rectangular area 200 coordinate positions wide and 300 coordinate positions high. The **at position** specification, which defines the graphical region's origin, is the lower left corner of the graphical region. For example:

```
graphical region Panel size 200 300 at position 100 50
```

This statement defines the graphical region Panel, which will be displayed as shown below. (In the illustration, the shaded area represents the graphical region; the shading is for illustration only, and does not appear on the screen.)



Remember that when you define a region's size, you are actually defining two characteristics: the region's viewport and the region's window. What is displayed in the region's window is like what you would see through a camera's lens. 7.3, "Action Statements for Changing Windows and Viewports in Graphical Regions" on page 7-47 shows how you can change the window and viewport, effectively "panning" and "zooming" a camera.

7.1.2.2 Ancestry in Graphical Regions

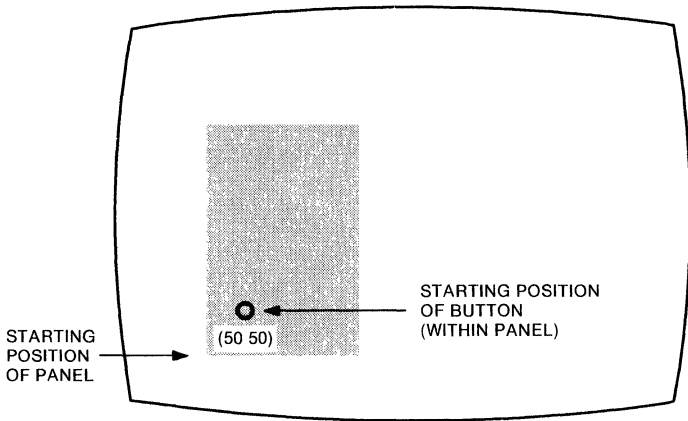
Graphical regions are often used as parents of other objects. For example, suppose you specify the following:

```
graphical region Panel size 200 300 at position 100 50
```

```
key Button at position 50 50 in Panel  
draw 100 100
```

Because the key named Button is defined as being **in** the graphical region named Panel, Button is the child of Panel. Panel is the parent, or the ancestor, of Button.

When a key is defined in this way, its coordinate positions are relative to the starting position of the graphical region. The starting position of Button is 50 50 within the Panel region, as shown in the example below:



7.1.2.3 Color in Graphical Regions

Unlike keys, graphical regions have three default colors: a foreground color, a background color, and a border color. In the graphical region definition below:

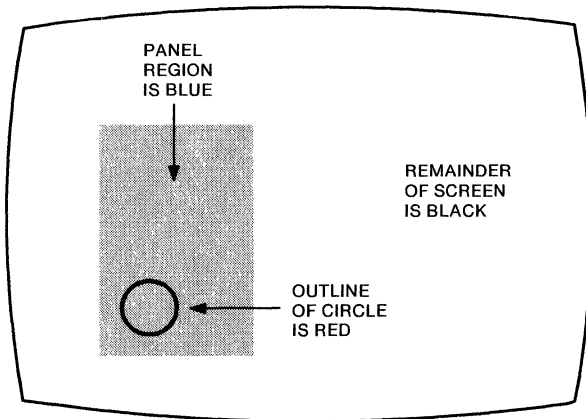
```
blue graphical region Panel size 200 300  
at position 10 50  
:
```

blue is the **background** color. This is the new default color of the rectangular area on top of which the graphical region's contents (drawing statements) are to be drawn. (If you do not specify a background color, the default is black.)

You can also specify the default color in which the graphical region's contents themselves are to be drawn. This is called the **foreground** color. (If you do not specify a foreground color, the default is white.) In the following example, red is the new default foreground color of the graphical region:

```
blue graphical region Panel size 200 300
  at position 100 50
  red foreground
  move 80 80
  circle radius 40
```

The contents of Panel—a circle—are drawn in red, the foreground color for the graphical region. This example displays the following:



7.2 Drawing Statements for Graphical Objects

7.2.1 Overview

The drawing statements described in this section can be used only in the contents of a key or a graphical region. If you specify one of these statements in the contents of any other object type, the drawing statement is not executed and EASEL OS/2 EE generates an error message.

Drawing statements for a key or a graphical region fall into several categories:

- Statements that only move the cursors:

```
move [by]
move to
```

- Statements that draw lines:

```
draw [by]
draw to
```

- Statements that draw patterns:

```
box
circle
polygon
wedge
ellipse
shape
```

- Statements that display or format text:

```
text [align]
text center
```

- Statements that display patterns that you have specified earlier in the program, with the **pattern** definition.

7.2.2 Placement of Drawing Statements

A drawing statement, or a series of drawing statements, describes the appearance of a key or a graphical region. The appearance of an object can be defined or changed in several places in the program. Drawing statements are valid in any of the following places in the program.

Object definitions. The initial appearance of an object is defined by the drawing statements that are specified in the contents of the object definition. For example:

```
invisible key Display at position 15 154
    box 90 22
```

Pattern definitions. A set of drawing statements that you might wish to repeat in several places in the program can be defined as a pattern. For example:

```
pattern Logo is
    draw 200 0
    draw 100 200
```

This pattern can then be used in a drawing statement in an object definition; for example:

```
key Corporate at position 10 35
    pattern Logo
```

Action statements within response definitions. You can add a new graphical object, or add to or replace the contents of an existing object, in an **add**, **add to**, or **add to [graphics]** action statement. For example:

```
response to BlueKey
    add to RedKey
        draw 100 50
```

When BlueKey is selected, the **draw** statement is added to the contents of RedKey.

7.2.3 Moving the Cursors

The statements **move [by]** and **move to** move the graphics and text cursors from their current positions to a new position, without drawing anything. The text cursor is moved to the same position as the graphics cursor, even if the two cursors were initially at different positions. The **move** statement uses the following model.

Model

```
move { [by] XOFFSET YOFFSET | to X Y }
```

The **move [by]** statement moves the graphics cursor from its current position to a new relative position specified by integer values representing the X and Y coordinates in the coordinate system of the object. For example, if the graphics cursor is at position 200 100 and you specify:

```
move 20 40
```


EASEL OS/2 EE moves the graphics cursor 20 positions to the right and 40 positions up, to position 220 140. EASEL OS/2 EE then moves the text cursor to this position.

The **move to** statement moves both cursors from their current positions to a new absolute position specified by integer values representing the X and Y coordinates in the coordinate system of the object. For example, if you specify:

```
move to 50 100
```

EASEL OS/2 EE moves the graphics and text cursors to coordinate position 50 100.

You can specify any integer value for the cursor position; for example:

```
# Width and Height are integer variables:  
move Width Height
```

or:

```
# xposition and yposition are built-in functions  
move to (xposition of A) (yposition of BlueKey)
```

7.2.4 Drawing a Line

The statements **draw [by]** and **draw to** draw a straight line from the current position of the graphics cursor to the specified position. These statements move the graphics and text cursors exactly as the **move** statements do, but they also draw lines from the old position to the new.

Model

```
[ [color] COLOR] draw  
  { [by] XOFFSET YOFFSET[, XOFFSET YOFFSET]... | to X Y [, X Y]... }
```

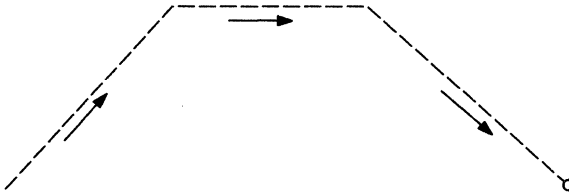
EASEL OS/2 EE begins at the current position of the graphics cursor. The cursor then moves the specified number of X and Y coordinates (in the coordinate system of the object), drawing a straight line to the new position. If additional sets of coordinates are specified in the statement, EASEL OS/2 EE continues to move the cursor and to draw additional lines. Each set of X and Y coordinates must be separated by commas. For example:

```
draw 50 50, 50 0, 50 -50
```

Here, the cursor moves 50 positions in the X direction and 50 positions in the Y direction, drawing a line from the current graphics cursor position to this new position. Then, from the new position, the cursor moves 50 positions in the X direction, continuing the line; because the Y coordinate is 0, the cursor does not move in the Y direction. Finally, the cursor moves 50

additional positions in the X direction and -50 positions in the Y direction, and the third line is drawn.

Regardless of the initial position of the graphics cursor, the line drawn by this example would look like:



The **draw to** statement draws a line beginning at the current position of the graphics cursor to an absolute position, moving the cursor to the specified coordinate positions. First, EASEL OS/2 EE draws a line from the current position to the new position indicated by the first X Y coordinates specified. Then, if additional sets of coordinates are specified in the statement, EASEL OS/2 EE continues drawing the line to the second set of coordinates, and so on. Again, each set of coordinates must be separated by commas. For example:

```
draw to 25 25, 100 100
```

Here, EASEL OS/2 EE draws a line from the starting position of the graphics cursor to position 25 25 in the object's parent. Then, EASEL OS/2 EE continues drawing from position 25 25 to position 100 100.

7.2.4.1 Colored Lines

You can specify a color for each line drawn. If you do not specify a color, EASEL OS/2 EE draws the line in the foreground color of the graphical object that you are defining. When you specify a color in the **draw** statement, this color overrides any other color specifications that might affect the color of the line. For example:

```
red key Button at position 50 50 in Panel
  draw 100 100
  move to (xmiddle of Button) (ymiddle of Button)
  blue circle radius 10
```

Here, the line drawn to position 100 100 does not have a specified color, so it is drawn in red, the foreground color of Button. But the color specification for the circle overrides the foreground color so it is drawn in blue.

You can specify a string or integer value in place of the color name; for example:

```
color ColorA draw to 25 25, 50 50
```

Both of these lines are drawn in the color stored in the variable (or constant), ColorA.

7.2.5 Drawing a Box

The **box** statement draws a rectangle, one pixel wide. You define the size of the desired box by specifying relative X and Y coordinates (in the coordinate system of the object in which the box is being drawn). EASEL OS/2 EE positions the lower left corner of the box at the current position of the graphics cursor. The position of the opposite corner (upper right) is determined by the X and Y specifications.

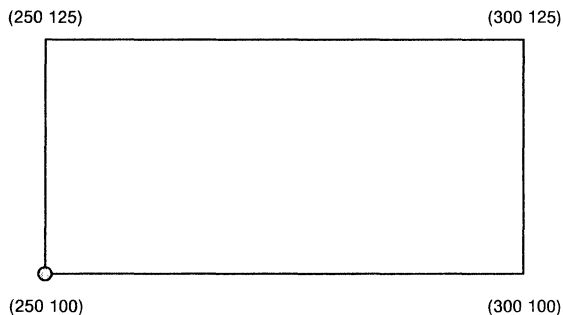
Model

```
[solid] [ [color] COLOR] box XOFFSET YOFFSET
```

After the box is drawn, the graphics cursor has returned to its initial position, and the text cursor is moved to this position. For example, suppose that you specify:

```
key KeyA at position 250 100
  box 50 25
```

The position specification (250 100) defines the origin of this key. (The graphics cursor is initialized to this position.) EASEL OS/2 EE begins drawing the box at that position. The box is shown in the following figure.



You can specify any integer value for the size; for example:

```
# Length and Height are integer variables or constants:  
solid red box Length Height
```

```
# xsize and ysize are built-in functions  
solid red box (xsize of TwinBox) (ysize of TwinBox)
```

7.2.5.1 Colored and/or Solid Box

You can color the perimeter of a box by specifying a color value in the **box** statement. For example:

```
red box 100 125
```

draws the perimeter of the box in red.

If **solid** is specified, EASEL OS/2 EE fills the box with the requested color (or with the default color, if none is specified). For example, the statement:

```
solid red box 100 125
```

draws a box and fills it with red.

7.2.6 Drawing a Polygon

The **polygon** statement, shown in the following model, draws a convex polygon (any closed figure bounded by straight lines). You define the polygon's size and shape by specifying relative X Y offset positions (in the coordinate system of the object in which the polygon is being drawn). This is like drawing a series of lines with the **draw** statement, with two important differences: all polygons are **solid** (filled), and EASEL OS/2 EE always creates a closed figure from the sets of coordinates specified.

Model

```
[ [color] COLOR] polygon XOFFSET YOFFSET[, XOFFSET YOFFSET]
```

Consider the following example, which draws a triangle:

```
key KeyA at position 250 100  
  polygon 10 10, 10 -10, -20 0
```

The polygon in KeyA draws a line 10 coordinates to the right of and 10 coordinates up from the current position of the graphics cursor (which is at position 0 0 in KeyA). Then EASEL OS/2 EE draws a line 10 coordinates to the right and 10 coordinates down from the new position. Finally, EASEL OS/2 EE draws a line 20 coordinates to the left of and 0 coordinates up or down from the last new position. Since this creates a closed shape, EASEL

OS/2 EE needs to do nothing more except fill the polygon with the color specified (or the default color, if none is specified).

If the polygon is defined as follows:

```
key KeyA at position 250 100
  polygon 10 10, 10 -10
```

the exact same figure is drawn, because EASEL OS/2 EE automatically closes the figure, drawing a line from the last specified position back to the origin.

Wherever possible, use the **polygon** statement instead of the **shape** statement, for faster printing and screen drawing.

7.2.6.1 Colored Polygon

You can color the polygon by specifying a color value in the **polygon** statement. For example:

```
red polygon 10 10,10 -10
```

or:

```
color NewCol polygon 10 10,10 -10
```

7.2.7 Drawing a Circle

The **circle** statement, shown in the following model, draws a circle. You indicate the size of the circle by specifying its radius.

Model

```
[solid] [ [color] COLOR] circle [radius] RADIUS
```

For example:

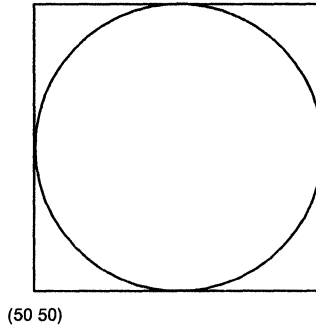
```
circle radius 20
```

draws a circle with a radius (or distance from the center) of 20 coordinate positions. EASEL OS/2 EE automatically draws the circle centered at the current position of the graphics cursor. The final position of the graphics cursor is the same as its initial position. The text cursor is moved to that position.

The following example draws a circle inside a box:

```
key BoxCircle at position 50 50
  box 300 300           # Draw a square box.
  move 150 150          # Move to center of box.
  circle radius 150      # Draw a circle touching
                        # the box.
```

This is shown below:



7.2.7.1 Colored and/or Solid Circle

You can color a circle by specifying a color value in the **circle** statement. For example:

```
red circle radius 30
```

draws the perimeter of the circle in red.

If **solid** is specified, EASEL OS/2 EE fills the circle with the requested color (or with the default color, if none is specified). For example, the statement:

```
solid red circle radius 30
```

draws a circle and fills it with red.

7.2.8 Drawing an Ellipse

The **ellipse** statement draws an ellipse. You define the size and shape of the ellipse by specifying integer values representing the ellipse's major radius and minor radius, respectively. EASEL OS/2 EE draws the ellipse centered at the current position of the graphics cursor. The final position of the graphics cursor returns to its initial position, and the text cursor is moved to that position.

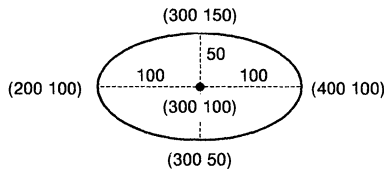
Model

```
[solid] [ [color] COLOR] ellipse
      major [radius] MAJOR minor [radius] MINOR
```

The major radius is parallel to the X axis (horizontal), and the minor radius is parallel to the Y axis (vertical). Either radius can be larger than the other, or the two radii can be equal. Suppose that you specify:

```
key Go at position 300 100
  ellipse major 100 minor 50
```

EASEL OS/2 EE draws an ellipse in which the radius from the center in the X direction is 100 positions; the total width is 200 positions. The radius from the center in the Y direction is 50 positions; the total height is 100 positions. This is shown in the example below:



Note that if the values specified for the major and minor radii are the same, as in the following:

```
ellipse major 20 minor 20
```

the resulting ellipse is the same as a circle, with a radius of 20 (assuming the display has square pixels).

7.2.8.1 Colored and/or Solid Ellipse

You can color the ellipse by specifying a color value in the **ellipse** statement. For example:

```
red ellipse major 50 minor 40
```

draws the perimeter of the ellipse in red.

If **solid** is specified, EASEL OS/2 EE fills the ellipse with the requested color (or with the default color, if none is specified). For example, the statement:

```
solid red ellipse major 50 minor 40
```

draws an ellipse and fills it with red.

7.2.9 Drawing a Wedge

The **wedge** statement, shown in the following model, draws a wedge, which is a common graphic in pie charts. You define the size and shape of the wedge by specifying the radius of the wedge, and the absolute starting and ending degrees (angle).

Model

```
[solid] [ [color] COLOR] wedge [radius] RADIUS  
from DEG1 degrees to DEG2 degrees
```

EASEL OS/2 EE starts drawing the wedge at the current position of the graphics cursor. EASEL OS/2 EE first draws the arc, starting at 3:00 o'clock on the circle of which the wedge is a part, and proceeds counter-clockwise. After the wedge is drawn, the position of the graphics cursor is the same as its initial position, and the text cursor is moved to that position. In the following example:

wedge radius 100 from 5 degrees to 90 degrees

EASEL OS/2 EE draws a wedge in which the radius is 100 coordinates; it begins at 5 degrees, and ends at 90 degrees, as shown below:



Note that if the values specified for the beginning and ending degrees are the same, as in the following:

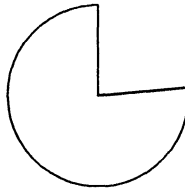
wedge radius 100 from 5 degrees to 5 degrees

the resulting wedge is a solid circle with a radius of 100.

You can specify the degrees in either order (from lowest to highest, or from highest to lowest). For example, if you specify:

wedge radius 100 from 90 degrees to 5 degrees

EASEL OS/2 EE draws the following:



You can specify any integer value for the degrees. If you specify a number higher than 360 or lower than 0, EASEL OS/2 EE adds or subtracts the 360 degree increments as necessary to fit the value within 360 degrees.

7.2.9.1 Colored and/or Solid Wedge

You can color the wedge by specifying a color value in the **wedge** statement. For example:

```
red wedge radius 100 from 5 degrees to 90 degrees
```

draws the perimeter of the wedge in red.

If **solid** is specified, EASEL OS/2 EE fills the wedge with the requested color (or with the default color, if none is specified). For example, the statement:

```
solid red wedge radius 100 from 5 degrees to 90 degrees
```

draws the wedge and fills it with red.

7.2.9.2 Limitations in Drawing Wedges

A wedge cannot be specified within a **boundary** statement.

7.2.10 Drawing Closed Figures (Shapes)

With the **shape** statement, you can create intricate images, such as a company logo. Although you can define many images by specifying any of the statements described thus far, the **shape** statement provides you with additional graphics capabilities. For example, you can draw a shape in one color and fill it with another color, fill an exterior boundary with color and create "holes," or draw a shape that contains arcs.

This section introduces two types of drawing statements that are used *only* within boundaries of shapes. The **draw arc** statement allows you to specify, as part of a boundary, an arc. An arc is a curved line, which is a segment of a circle. The **move arc** statements allow you to move the graphics cursor in an arc without drawing an arc.

```
[solid] [ [color] COLOR] shape
  [ [ [color] COLOR] border]
  boundary
    [MOVE_STATEMENT]
    DRAWING_STATEMENT
    [ DRAWING_STATEMENT | MOVE_STATEMENT ]...

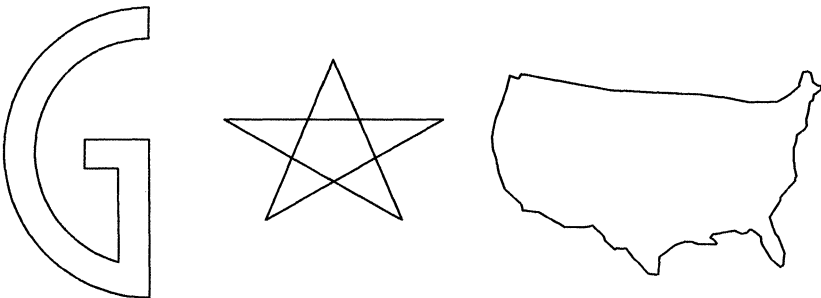
[boundary
  [MOVE_STATEMENT]
  DRAWING_STATEMENT
  [ DRAWING_STATEMENT | MOVE_STATEMENT ]... ]...

end shape
```

7.2.10.1 Boundaries in Shapes

A **shape** statement must contain at least one **boundary** specification. A boundary is a closed figure; each boundary consists of one or more drawing statements that describe the outline of a closed figure. A boundary can have straight lines, specified by the **draw** statements. It can also be rounded, using the **circle**, **ellipse**, or **draw arc** statements. A single **shape** statement can have a mixture of any of these statements.

In a single **shape** statement you can define the boundaries of multiple figures. You can create nested figures, figures that overlap one another, and figures that do not touch one another. Boundaries can be as simple or as complex as you wish. The following are boundaries:



Since a boundary is a closed figure, a boundary specification must result in either a closed figure or a figure that EASEL OS/2 EE can close. EASEL OS/2 EE closes any unclosed boundary by drawing a single straight line

from the ending coordinate position back to the beginning coordinate position of the boundary.

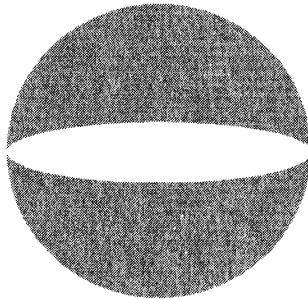
The first drawing statements in a boundary specification can be **move** or **move arc** statements, to position the graphics cursor for use by subsequent statements. You can only use these statements *prior* to the first **draw** statement; they cannot appear later in the boundary, since they would interrupt the continuous lines of the boundary (and EASEL OS/2 EE would be unable to make a closed figure).

Note: Although **wedge** and **polygon** statements result in closed figures, you cannot specify them in a **boundary** statement.

The **circle** and **ellipse** statements may be specified in boundary specifications. A boundary that contains either of these statements is already fully closed; it cannot have additional drawing statements other than the optional **move** or **move arc** statements to position the graphics cursor. For example, the following statement:

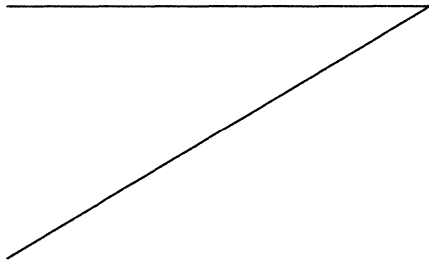
```
solid shape
  boundary circle radius 100
  boundary ellipse major 100 minor 40
end shape
```

is a **shape** statement with two boundary specifications. The first boundary is a circle, and describes an outer boundary. The second boundary is an ellipse, and describes a hole which is not filled (colored). This shape is shown below:



If a boundary consists of only straight lines (that is, **draw** statements), the statements must describe at least two lines of the boundary.

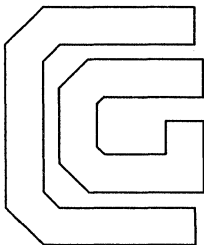
A single **draw arc** statement can be a complete boundary specification because EASEL OS/2 EE can draw a straight line between the starting point and the endpoint of the arc. The figure below is a **boundary** specification.



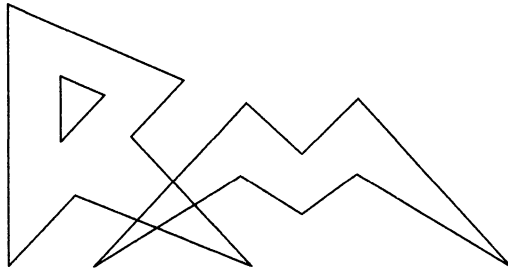
Because this is not a closed figure, EASEL OS/2 EE automatically closes the figure when creating the boundary, as shown below:



You can specify any number of boundary specifications between the **shape** and the **end shape** keywords. The following example shows two boundaries that do not touch one another:



The example below has three boundary specifications:



In this shape, one boundary completely contains another and is overlapped by a third.

The boundaries that you specify can be in any order. In the example above, you could have first specified the boundary that is displayed on the right (the “M”), then the hole in the “R”, and finally the “R” itself.

7.2.10.2 Graphics Cursor in Shapes

The position of the graphics cursor is updated after each drawing statement within the boundary specification, just as it is after any drawing statement. However, after each boundary has been defined, the graphics cursor is restored to the position it had at the beginning of the **shape** statement. Therefore, each boundary that you describe in a single **shape** statement begins at the same position, unless you reposition the graphics cursor by specifying a **move** or a **move arc** statement before the boundary’s other drawing statements.

Restoring the graphics cursor in this way between a **shape** statement’s boundary specifications frees you from making errors in specifying the positions of multiple boundaries in a shape.

The **shape** statement shown below consists of several boundary specifications.

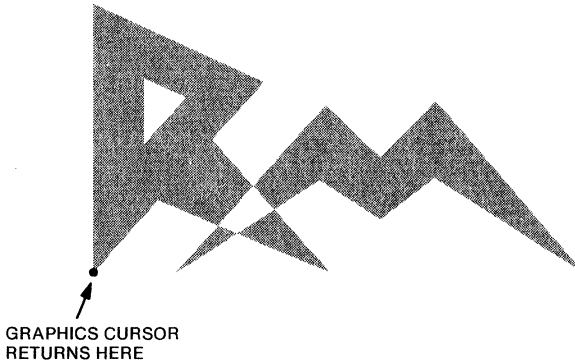
```
solid magenta shape
  boundary                # Exterior of letter R
    draw 0 200
  :
  boundary move to 30 160 # Hole of letter R
    draw 20 0
  :
```

```

boundary move to 40 0   # Letter M
draw 30 50
:
end shape

```

After each boundary is drawn, the graphics cursor is restored to its starting position. The first drawing statement in the second and third boundary specifications begins with a **move** statement to reposition the graphics cursor for the new boundary.



7.2.10.3 Drawing an Arc of Specified Degrees

The **draw arc** statement draws, within a boundary, an arc of the specified number of degrees, or an arc to a specified endpoint. The arc always begins at the current position of the graphics cursor. To draw an arc of a specified number of degrees, use the following model.

Model

```

draw arc DEGREES degrees [ counter clockwise | clockwise ]
center [ by | to | at ] XCENTER YCENTER

```

For example, the following statement:

```
draw arc 30 degrees
```

draws an arc 30 degrees (counterclockwise, by default) from the current position of the graphics cursor.

The **center** specification specifies the center of the circle of which the arc is a portion. Therefore, all points along the arc are equidistant from the center point. The center can be specified as an absolute position in the coordinate

system of the object (the synonymous **center at** or **center to** specifications), as in the following examples:

```
draw arc 30 degrees center at 0 0
```

or

```
draw arc -45 degrees center to 50 50
```

The **center at** and **center to** specifications are synonymous. They describe the center as an absolute position in the coordinate system of the object.

Or, the center can be specified as a position relative to the start of the arc, which is at the current position of the graphics cursor (the **center [by]** specification); for example:

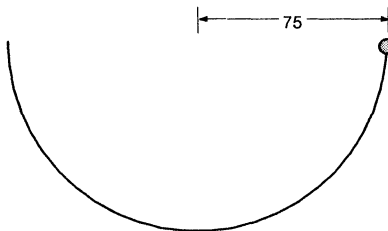
```
draw arc 180 degrees center by 75 0
```

You can optionally specify the direction in which the arc is to be drawn. The direction of the arc is specified by the keywords **clockwise** or **counter clockwise**. If you omit either specification and specify a positive integer value for the degrees, the arc is drawn counterclockwise. If you omit either specification and specify a negative integer value for the degrees, the arc is drawn clockwise.

The **center [by]** specification describes the center as an offset from the beginning position of the graphics cursor. In the following example:

```
draw arc 180 degrees center by 75 0
```

EASEL OS/2 EE establishes the center by adding the specified X and Y offsets to the starting position of the graphics cursor. This statement results in a semicircle with a diameter of 150, as shown below:



This figure shows the position of the graphics cursor after the arc is drawn.

You must provide a **center** specification for the first arc in the first **shape** statement in a graphical object. Thereafter, if you specify a series of **move arc** and **draw arc** statements in a single graphical object, and if each arc has the same center, you do not need to provide additional **center** specifications. If you do not specify a **center** specification in one of these **move arc**

or **draw arc** statements, the center is assumed to be the same as the center of the previous arc in that graphical object.

7.2.10.4 Drawing an Arc to a Specified Endpoint

Within a boundary, you can also draw an arc from the current position of the graphics cursor to a specified endpoint. To do this, use the following model.

Model

```
draw arc { by | to } XEND YEND [ counter clockwise | clockwise ]
      center [ by | to | at ] XCENTER YCENTER
```

As shown in the model, the endpoint of the arc can be specified as an absolute position in the coordinate system of the object (the **draw arc to** statement, or a position relative to the beginning position of the graphics cursor (the **draw arc by** statement).

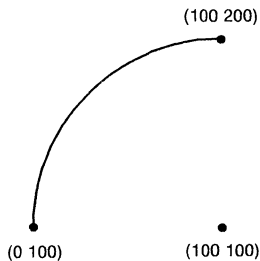
The **center** specification in these statements is the same as it is when you use degrees in the **draw arc** statement. That is, the **center at** or **center to** specifications describe the center as an absolute position in the coordinate system of the object. The **center by** specification describes the center as an offset from the beginning position of the graphics cursor.

You can optionally specify the direction in which the arc is to be drawn. The direction of the arc is specified by the keywords **clockwise** or **counter clockwise**. If you omit both of these specifications and provide a positive integer value for the endpoints, the arc is drawn counterclockwise.

In the **draw arc to** statement, the specified X and Y values are absolute positions in the coordinate position of the object. If the graphics cursor is at position 0 100, the statement below:

```
draw arc to 100 200 center at 100 100
```

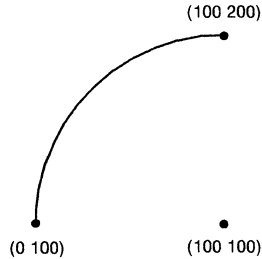
draws the following arc:



In the **draw arc by** statement, the endpoint of the arc is the position created by adding the specified X and Y values to the current coordinates of the graphics cursor. For example, if the graphics cursor is at position 0 100, the statement:

```
draw arc by 100 100 center at 100 100
```

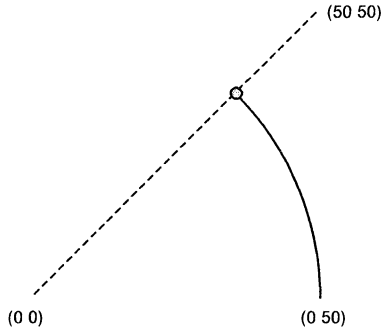
draws an arc to position 100 200, as shown below:



Sometimes it is not possible to draw an arc exactly as specified. When EASEL OS/2 EE draws an arc, each point along the arc is equidistant from the point defined as the arc's center. When you draw an arc to a specified endpoint using the **to** specification and if the starting point and the endpoint of the arc are not the same distance from the center, the arc cannot touch both the starting and ending points and still be circular. In this case, the arc is drawn with the starting point and the center exactly as specified, and the endpoint of the arc is drawn as close as possible to the specified endpoint, while preserving the circularity of the arc. For example, in the following statements:

```
move to 0 50  
draw arc to 50 50 center at 0 0
```

the starting point of the arc is at position 0 50. The center is at position 0 0. The arc is drawn with the endpoint as close as possible to position 50 50 while still maintaining the rule that each point on the arc is the same distance from the center as each other point. The arc that results from these statements is shown in the following figure.



The endpoint of an arc drawn in this way always lies at a point along the radius from the center to the specified endpoint.

As discussed above, a **draw arc to** or a **draw arc by** statement that does not have a **center** specification assumes as its center the center specified for the previous arc in that graphical object.

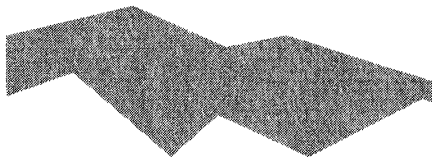
7.2.10.5 Moving in Arcs

You can move the graphics cursor in arcs, in either degrees or specified endpoints, just as you draw arcs. The only difference is that the **move arc** statement moves the graphics cursor from its current position to a new position without drawing an arc.

7.2.10.6 Solid Shapes

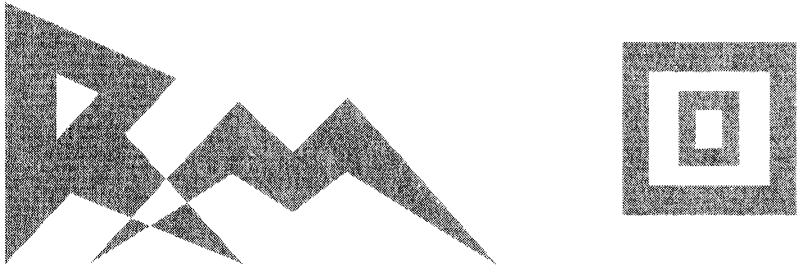
If you specify the keyword **solid** in a **shape** statement, EASEL OS/2 EE fills the shape with color. If **solid** is not specified, only the boundaries are drawn.

In the case of a solid shape with a single boundary whose drawing statements do not cross each other, EASEL OS/2 EE draws a single boundary and fills it with color, as shown below:



In the case of multiple boundary specifications in a single **shape** statement, EASEL OS/2 EE draws all the boundaries and then fills the shape so that no

boundary has filling on both sides. Each of the following shapes has at least two boundaries, and is filled as shown:



7.2.10.7 Color in Shapes

The **shape** statement allows you to specify a color in which the boundaries are drawn, and a different color in which the shape is filled. If you do not specify a color for the shape, its color is inherited from the pattern or from the foreground of the graphical object of which it is a part. (If no color is specified anywhere, the shape is drawn in white.)

The shape's color is used to draw the boundaries of the shape. If the keyword **solid** is specified, the shape is also filled in the same color.

EASEL OS/2 EE draws all boundaries in a single **shape** statement in the same color. You cannot specify one color for one boundary and another color for another boundary.

If you wish to display shapes that are drawn in different colors, you can specify two or more separate **shape** statements.

7.2.10.8 Border in Shapes

The **border** specification allows the boundaries of the shape to be drawn in a color that is different from the color with which the boundaries are filled. You specify a border color with the keyword **border** preceded by a color value. For example:

```
solid blue shape
  red border
  boundary ...
  :
end shape
```

This shape is drawn in red and filled with blue. If the **border** (and border color) is omitted, the shape is both drawn and filled with blue. (The only logical use of **border** is when **solid** is also specified.) Note that the boundaries are drawn in the border color even if **solid** is not specified; for example:

```
blue shape
  red border
  boundary ...
  :
end shape
```

draws *every* part of this shape in red—*none* are drawn in blue.

7.2.11 Defining and Referencing Patterns

In addition to the drawing statements that create built-in patterns (such as the box, the circle, and the ellipse discussed above), you can also define your own patterns.

A **pattern** is a set of drawing statements. Defining and referencing patterns can significantly reduce the amount of time you spend in creating graphical objects. Once you have defined a pattern, you can specify an entire series of drawing statements in the definition of a graphical object simply by referencing the name of the pattern, rather than by repeating all of the drawing statements in the contents of each graphical object that will contain them. A single object definition can have many pattern references, as well as other drawing statements. (Textual regions do not contain graphics, and therefore cannot contain patterns.)

You use the **pattern is** statement to define a pattern as a set of drawing statements. You use the **pattern** statement to reference a pattern that you have defined.

Modifications to the display of a basic pattern can be made when the pattern is referenced. For example, the color of a pattern, or the color of one or more of the drawing statements of a pattern, can be specified in a pattern reference. A pattern can be displayed as larger or smaller than the size in which it was defined, and it can be rotated.

You can reference a pattern in the definition of another pattern. Patterns can be nested to any depth.

7.2.11.1 Defining a Pattern

The **pattern is** statement defines a pattern. A pattern consists of one or more drawing statements, as shown in the following model. The definition must appear before any reference to it.

Model

pattern PATTERN_NAME is

[DRAWING_STATEMENT]...

You can use only those drawing statements that are legal for graphical objects; textual region drawing statements are not permitted in patterns. For example:

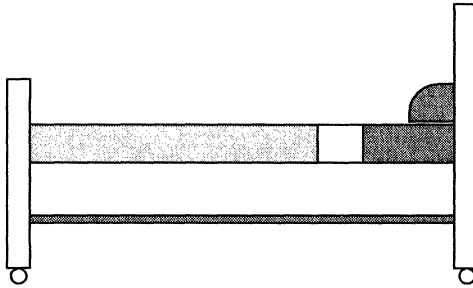
```
pattern GraphAxes is
  black draw -300 0, 0 300
  move 0 -300
  red draw 0 -300
```

When you define a pattern, you specify the properties of the pattern that will be common to all of the graphical objects that reference it. For example, you might define the following pattern:

```
screen size 640 350
```

```
pattern HospitalBed is
white box 6 50          # Footboard
move 3 -2
white circle radius 2   # Footboard wheel
move 113 2
white box 6 80          # Headboard
move 3 -2
white circle radius 2   # Headboard wheel
move -113 17
solid white box 110 1   # Support bar
move 0 15
solid box 70 10         # Blanket
move 70 0
white box 15 10         # Fold of sheet
move 15 0
solid white box 25 10   # Sheet
move 10 11
solid white shape       # Pillow
boundary
  draw 15 0
  draw 0 10
  draw -5 0
  draw arc 90 degrees
    center by 0 -10
end shape
```

This pattern is displayed as shown below:



7.2.11.2 Referencing a Pattern

You use the **pattern** statement to reference a previously defined pattern. A pattern reference is a drawing statement. It can be specified anywhere in the program that a drawing statement can be specified, except as part of a **boundary** specification in a **shape** statement. Once a pattern has been defined, it can be referenced in any number of object definitions. The **pattern** reference statement uses the following model.

Model

```
[ [color] COLOR] [scale [ by | to ] NUMERATOR:DENOMINATOR]  
[rotate [ by | to ] DEGREES] pattern PATTERN_NAME
```

The following example references the pattern HospitalBed (which was defined in 7.2.11.1, “Defining a Pattern” on page 7-30):

```
yellow key Ward2 at position 300 150  
  pattern HospitalBed
```

Regardless of the other properties of the key Ward2, HospitalBed will always consist of the frame of the bed (a headboard, a footboard, a support bar, and wheels), a blanket, a sheet, the fold of the sheet, and a pillow.

You could have ten separate graphical objects, each representing one bed in a ten-bed ward. The current occupancy status of each bed could be represented by the color of the bed’s blanket—for example, a yellow blanket for an occupied bed and a blue blanket for an empty bed. Therefore, the blanket would be drawn in the color specified in the pattern reference. (If the pattern reference does not specify a color, the blanket inherits its color from the pattern reference or object definition.) Also, the color of a bed’s blanket could be changed when the occupancy status of the bed changes, in an action statement (if no color was specified when the pattern was referenced); for example:

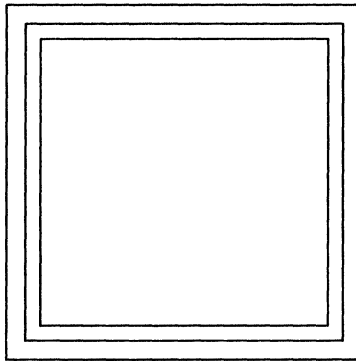
```
response to Bed1
  make Bed1 red
```

7.2.11.3 Reference Point of a Pattern

As EASEL OS/2 EE executes each drawing statement in a pattern, it moves the graphics cursor and the text cursor of the object that contains the pattern reference. The cursors are moved exactly as they would be if the drawing statements of the pattern were specified in the object definition. In the example below:

```
pattern Cubes is
  box 200 200      # Outermost box
  move 10 10
  red box 180 180  # Middle box
  move 10 10
  green box 160 160 # Innermost box
  move 80 80       # Move to center
```

the pattern, Cubes, consists of six drawing statements that describe a set of three nested boxes, as shown below:



The final drawing statement positions the text cursor and the graphics cursor in the center of the innermost box. The key defined below references this pattern.

```
key Blue at position 100 200
  blue pattern Cubes
  text center "Blue Box"
```

When EASEL OS/2 EE draws the Blue key on the display, it begins the pattern Cubes at position 100 200. The position at which EASEL OS/2 EE begins to draw a pattern is called the *reference point* of the pattern.

After this pattern has been executed, the cursors are in the center of the innermost box. The final drawing statement centers the text string, "Blue Box," horizontally and vertically.

The pattern's reference point is the position of the graphics cursor at the time that the pattern is referenced. Each pattern reference establishes a reference point for that pattern within that object; for example:

```
key First at position 300 150
  move 100 -50
  pattern Star
```

This object definition establishes the reference point of the pattern, Star, as position 100 -50 in the coordinate system of the key First, because that is the position of the graphics cursor at the time the pattern is referenced. In the object definition below, the reference point of the pattern is position 50 0 in the coordinate system of the key Second.

```
key Second at position 200 200
  draw 50 100, 0 -100
  pattern Star
```

The reference point of a pattern is also used when the pattern is scaled or rotated. Pattern scaling and rotation in 7.2.11.8, "Scaling a Pattern" on page 7-36 and 7.2.11.9, "Rotating a Pattern" on page 7-37.

7.2.11.4 Coordinates of a Pattern

The coordinates referred to in a pattern are in the coordinate system of the graphical object that references the pattern. When an object contains a pattern reference, the drawing statements in the pattern begin at the current position of the object's cursors, and all movement within a pattern is performed in the coordinate system of the object. Therefore, when you specify coordinates in the drawing statements of a pattern, it is advisable to specify relative movement (e.g., **draw [by]**) rather than absolute movement (e.g., **draw to**). For example:

```
draw by 100 100
```

draws a diagonal line, 100 coordinate positions to the right and 100 coordinate positions upward, beginning at the position of the graphics cursor. The angle and the length of this line are always the same, regardless of the graphics cursor position at the time the pattern is referenced.

Although EASEL OS/2 EE allows you to specify absolute movement in a **pattern** statement, such as:

```
draw to 100 100
```

This draws a line beginning at the graphics cursor and ending at coordinate position 100 100 of the object that references the pattern. The appearance

of the line therefore depends on the position of the graphics cursor relative to the origin of the object that contains the pattern reference, and the angle and the length of the line will differ from one reference to another.

7.2.11.5 Nested Patterns

Any pattern definition can contain a reference to another pattern. A pattern that references another pattern is a parent of that pattern. A child pattern can, in turn, reference one or more other patterns. There is no practical limit to the number of generations of patterns (levels of nesting) that you can establish in this way. A pattern cannot, however, reference itself.

7.2.11.6 Color in a Pattern

Color can be specified for individual drawing statements in a pattern definition, or for the entire pattern in a pattern reference. You cannot specify color for the entire pattern in a pattern definition.

When a pattern reference specifies a color, any drawing statement that is part of the pattern and does not have its own color specification is drawn in the color specified in the pattern reference. In the example of nested boxes:

```
pattern Cubes is
  box 200 200          # Outermost box
  move 10 10
  red box 180 180      # Middle box
  move 10 10
  green box 160 160    # Innermost box
  move 80 80          # Move to center
```

the drawing statements that display the two inner boxes have color specifications; the middle box is red and the innermost box is green whenever the pattern Cubes is displayed. The drawing statement that displays the outermost box does not have a color specification. When this box is drawn, it is drawn in the color specified in the pattern reference or the foreground of the referencing object, if no color specification is given. For example:

```
aqua pattern Cubes
```

draws the box in aqua, whereas:

```
pattern Cubes
```

draws the outermost box in the foreground color of the graphical object whose contents contain the pattern reference.

If the graphical object that contains this pattern reference has no color specification, the outermost box is drawn in white:

```
key Shapes at position 300 200
  pattern Cubes
```

If a color has been specified for a particular drawing statement in a pattern definition, it cannot be changed for that drawing statement in any pattern reference.

7.2.11.7 Font in a Pattern

A font can be specified for individual drawing statements in a pattern definition. For example, the following pattern definition specifies a font for a particular drawing statement:

```
pattern Text is
  font "large" text center "Hello"
  text align "Good Morning"
```

Whenever this pattern is referenced, the text string, "Hello", is displayed in the "large" font. Any drawing statement that specifies a text string without a font specification, such as "Good Morning" in the above example, is displayed in the default font for the program. If no default program font is specified, the **medium** font is used.

Note that a font cannot be specified in a pattern reference.

7.2.11.8 Scaling a Pattern

When you reference a pattern in an object definition, you can enlarge or shrink it with the **scale** specification. You specify the enlargement factor by providing positive integer values representing the numerator and denominator; EASEL OS/2 EE determines the scale factor of the pattern by dividing the numerator by the denominator.

If the *scale factor* is less than 1, the pattern is displayed smaller. For example, specifying:

```
scale 1:2
pattern StopSign
```

displays the pattern at half the size of its definition. If the scale factor is greater than 1, the pattern is displayed larger. For example, specifying:

```
scale 2:1
pattern Butterfly
```

displays the pattern at twice the size of its definition.

In the example of nested cubes, defined as:

```

pattern Cubes is
  box 200 200      # Outermost box
  move 10 10
  red box 180 180  # Middle box
  move 10 10
  green box 160 160 # Innermost box
  move 80 80       # Move to center

```

the dimensions of the outermost box are 200 by 200. If you specify a **scale** statement, as shown below:

```

scale 1:2
  pattern Cubes

```

the pattern is displayed at half the size of its definition. The dimensions of the outermost box are 100 by 100.

Both the original pattern and the scaled pattern begin at the pattern's reference point. A pattern is always drawn beginning at that point, regardless of the scale factors specified.

The scalings specified for a pattern is cumulative when patterns are nested. The specified scaling is relative to any scaling currently in effect for the pattern.

7.2.11.9 Rotating a Pattern

When you reference a pattern in an object definition, you can rotate it. Rotation is performed with respect to the pattern's *reference point*. Note that both the original pattern and the rotated pattern begin at the pattern's reference point. A pattern is always drawn beginning at that point, regardless of the specified rotation.

The **rotate** specification rotates a pattern by a specified number of degrees before drawing it. The degrees specification is an integer value, representing the number of degrees by which all points in the pattern are rotated about the pattern's origin. If the degrees value is positive, the pattern is rotated counterclockwise. If it is negative, the pattern is rotated clockwise.

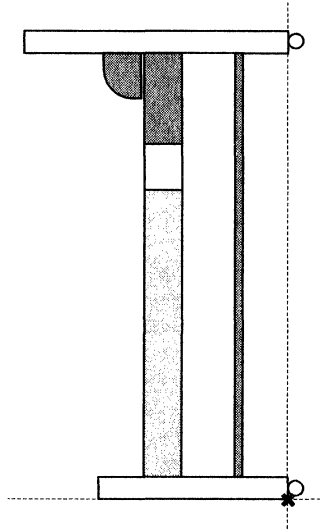
Rotation is specified in any number of degrees. For example, to rotate the HospitalBed pattern by 90 degrees counterclockwise, you would specify:

```

rotate 90
  pattern HospitalBed

```

The rotated pattern is shown below:



If you specify a number of degrees greater than 360, EASEL OS/2 EE subtracts 360 from the specified value and rotates the pattern by the remainder.

Take special care when rotating any patterns that contain a **box** statement; because a box is always drawn on the horizontal and vertical axes, a rotated box will cause unpredictable results unless you rotate it in 90-degree increments. EASEL OS/2 EE rotates the four corners of the box by the specified number of degrees and then redraws the smallest box that contains all four corners. For example:

```
key Before at 200 200
```

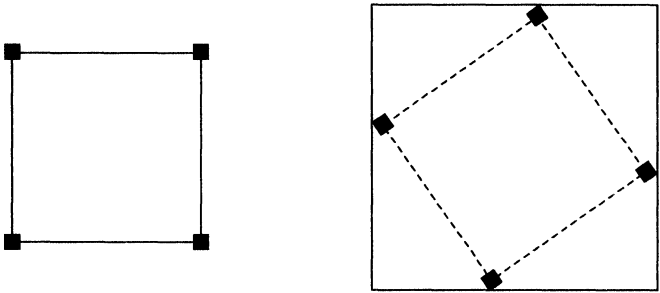
```
key Rotated at 400 200
```

```
pattern Box is  
  blue box 50 50
```

```
# Continues...  
response to ShowRotate  
  add to Before  
    pattern Box  
  add to Rotated  
    rotate 40 pattern Box
```

When **response to ShowRotate** is executed, the pattern Box is displayed in the key Rotated as shown below. The figure on the left shows the unrotated

pattern Box; the figure on the right shows how the pattern Box is rotated, and the new box drawn from the four corners of the rotated pattern.



If the pattern that you are rotating is an ellipse, rotation is approximate. EASEL OS/2 EE rotates the ellipse to the nearest 90 degrees.

The rotations specified for a pattern are cumulative when patterns are nested. The specified rotations are relative to any rotation currently in effect for the pattern.

7.2.11.10 Scaling Text

You can scale only text that is defined as having the **outline** attribute. When text is scaled, EASEL OS/2 EE moves the graphics cursor and the text cursor as specified by the **scale** specification. The new cursor positions dictate the starting position of a text string.

7.2.11.11 Rotating Text

You cannot rotate text; it is always shown horizontally, left to right.

7.2.11.12 Accumulation of Scaled and Rotated Patterns

One effect of nested patterns is that the specified scale factors or rotations can be cumulative. For example, if one pattern contains a reference to another pattern, and the reference scales the pattern, as shown below:

```
pattern A is
  scale 2:1 pattern B
```

then pattern A is pattern B at twice the size at which pattern B was defined. If you then specify:

```
pattern C is
  scale 2:1 pattern A
```

then pattern C is actually pattern B, doubled in size to become pattern A, then doubled in size again to become pattern C.

The following examples illustrate the effects of scaling and rotating patterns. Take particular notice of the position of the pattern's reference point and of any effects of accumulation of the **scale** and **rotate** specifications. These examples use the two patterns defined below:

screen size 1200 900

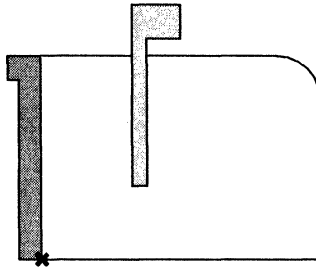
```
pattern Box is
  white shape          # Box
    boundary
    draw 300 0, 0 150
    draw arc 90 degrees center by -50 0
    draw -250 0, 0 -100
  end shape
  solid white shape    # Door
    boundary
    draw -10 0, 0 190, -10 0, 0 10, 20 0
  end shape
```

```
pattern Flag is
  solid red shape
    boundary
    draw 10 0, 0 160, 40 0, 0 40, -50 0
  end shape
```

The first pattern is a mailbox. The second pattern is the red flag seen on mailboxes. The pattern defined below contains a reference to each of these patterns:

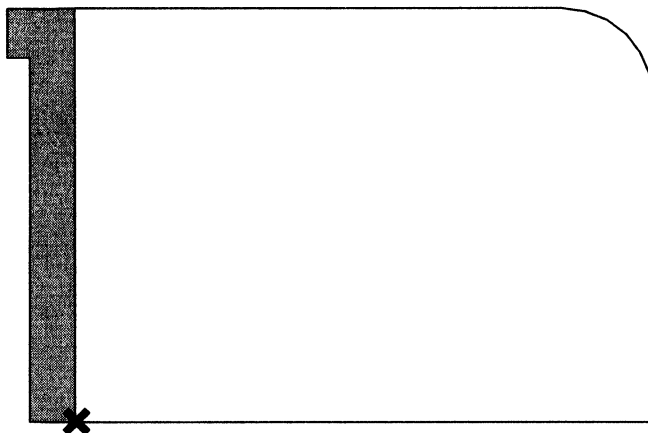
```
pattern Mailbox is
  pattern Box
  move 100 80
  pattern Flag
```

Mailbox is displayed as shown below. The reference point for the pattern is marked.



The following are illustrations of transformations on these patterns.

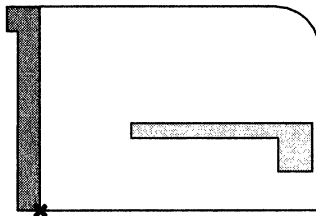
scale 2:1 pattern Box



pattern Box

move 100 80

rotate -90 pattern Flag



The following pattern definitions:

pattern BigBox is

scale 2:1

pattern Mailbox

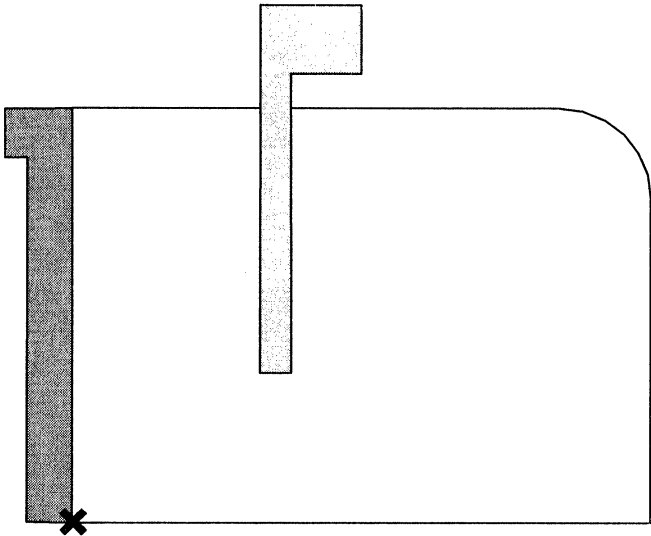
pattern TippedBox is

rotate 30

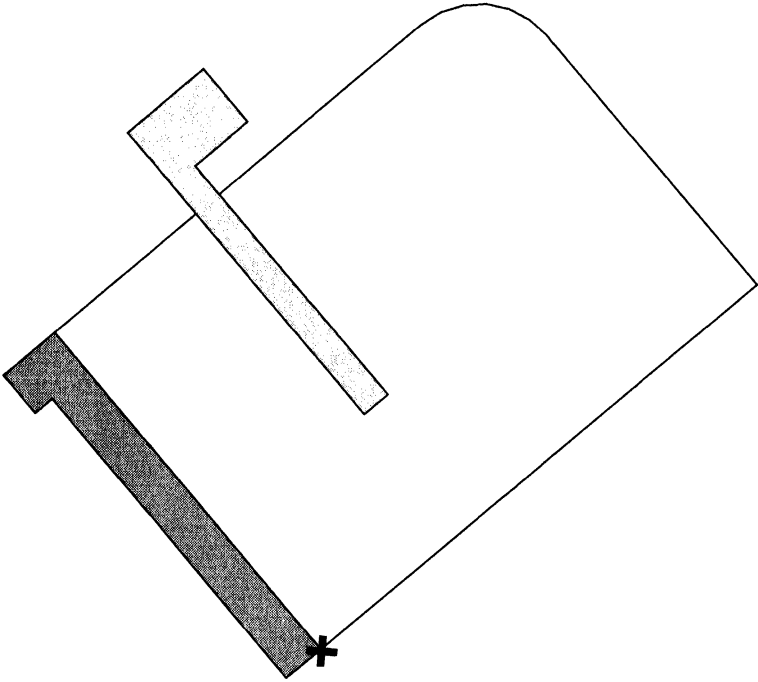
pattern BigBox

contain an example of a nested pattern. The first pattern, BigBox, consists of a scaled version of Mailbox. The second pattern, TippedBox, consists of a rotated version of BigBox.

BigBox looks like this:



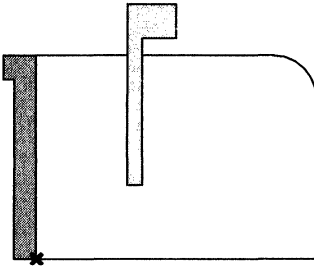
It is the pattern Mailbox shown at twice its size. When TippedBox is displayed, it is a rotated version of BigBox, as shown below:



If you then define:

```
pattern SmallBox is  
  scale 1:2  
  pattern BigBox
```

BigBox is Mailbox scaled to twice its size, and SmallBox is BigBox scaled to half its size. Thus, the pattern Mailbox is displayed in the size at which it was initially defined:



7.2.12 Displaying Text in Graphical Objects

The drawing statements in this category display text strings and control the formatting of text in graphical objects. These statements cannot be used in textual regions.

Each **text** statement is independent of all others; for example, any graphical **text** statement is independent of all object can have both centered and aligned text statements intermixed.

The **text** statement, shown below, allows you to display text (such as labels, lists, etc.) in a graphical object. When you display a text string, you can left-justify the text at the current position of the text cursor, or you can center the text with the text cursor position as its midpoint.

Model

```
[ [color] COLOR] [font FONT_NAME]
[text [ align | center ] ] TEXT_STRING
```

You can specify any character in the **text** statement. You can specify any string value, including string variables or string constants, as text; for example:

```
key Output at position 150 200
  move to 150 200
  text Message
```

You can specify any number of string values at one time. For example:

```
key SelectMe at position 300 150
  "Button " Button
```

displays the text "Button ", followed by the value of the string variable Button.

7.2.12.1 Displaying Image Text in a Graphical Region

When you display graphical objects in a graphical region's window, the image can be displayed larger or smaller, depending on the size and proportions of the viewport and window. However, you cannot manipulate image text in the same way. You cannot reduce or expand it to fill the viewport. (See 5.6.7, "Image Fonts" on page 5-49.)

7.2.12.2 Displaying Outline Text in a Graphical Region

You can display outline text in any graphical object. As the region is scaled, outline text reduces or expands. EASEL OS/2 EE draws as much of the outline text as possible. (See 5.6.8, "Scalable Outline Fonts" on page 5-50.)

7.2.12.3 Aligning Text

When a text string is aligned, either by default or because the keyword **align** is specified, EASEL OS/2 EE displays text left-justified at the current position of the text cursor. The lower left corner of the first character in the string is placed at the text cursor. In the example below:

```
key Example at position 200 300
  text align "This is alignment"
```

the text cursor and the graphics cursor are initially at position 200 300 in the screen. The text string, "This is alignment," will begin at that position.

After the text has been displayed, the text cursor is moved to the position immediately following the string, so that the next text displayed will be correctly positioned. The graphics cursor does not move.

The keywords **text align** can be omitted; for example:

```
key Ready at position 150 200
  "ALL SYSTEMS GO"
```

or

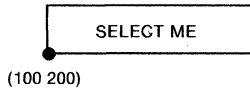
```
key Ready at position 150 200
  Message
```

7.2.12.4 Centering Text

When the keyword **center** is specified, EASEL OS/2 EE displays the text so that it is centered both horizontally and vertically at the current position of the text cursor. In the following example:

```
key Select at position 100 200
  box 150 100
  move to 75 50
  text center "SELECT ME"
```

EASEL OS/2 EE first draws a box that is 150 coordinate positions wide and 100 coordinate positions high, beginning at position 100 200 in the screen. The **move to** statement moves the text cursor (and the graphics cursor) to the center of the box. The **text center** statement then positions the text string, "SELECT ME," exactly in the center of the box. The Select key is shown in the following figure.



As with aligned text, the text cursor is moved to the bottom right of the last character cell in the string.

7.2.12.5 Font/Color of Text

You can specify the **font** in the **text** statement. For example, the statement:

```
font "large" text "Big Font"
```

displays the string, "Big Font", in the **large** font. If no font is specified, the text is displayed in the font currently in effect. You can also specify the color in which the text string is to be displayed. For example:

key Select...

```
red font "medium" text "Hello"
```

7.3 Action Statements for Changing Windows and Viewports in Graphical Regions

7.3.1 Introduction

Displaying the contents of a region's window in the region's viewport is like using a camera. You control what is visible in the camera's lens by aiming the camera; what you see through the lens is like the window of a region. The viewport is like a snapshot; it is a picture of the window. The following sections describe the effects of altering the size and position of a window and viewport.

You might define a graphical region called Blackboard and a key called Round, as shown below. Round is defined as a child of Blackboard.

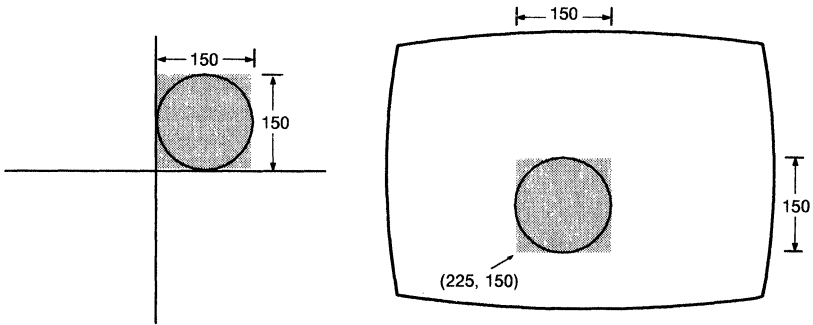
```
screen size 640 350
```

```
graphical region Blackboard size 150 150  
  at position 225 150
```

```
key Round at position 75 75 in Blackboard  
  circle radius 75
```

The viewport of the graphical region Blackboard is a rectangular area whose size is 150 coordinate positions wide and 150 coordinate positions high. The lower left corner of the viewport is at position 225 150 in the screen. Blackboard's window is the same size, and is initialized at position 0 0 in the graphical region.

The key Round consists of a single circle. The origin of the circle (75 75) places it in the center of Blackboard's window. Since the radius of Round is 75 coordinate positions, it touches the outermost positions of Blackboard's window at four places. In the figure below and in the remaining figures in this section, the diagram on the left shows Blackboard's window positioned in the graphical region, and the diagram on the right shows Blackboard's viewport positioned on the display. (The examples are not cumulative; each starts with the original definitions of Blackboard and Round.)



The entire contents of a region's window appear in the region's viewport, regardless of any distortion that may result. This means that if you change the proportions of either the viewport or the window, but do not change the proportions of the other, then the image that appears in the viewport is distorted.

It is important to remember that throughout all of the alterations in window and viewport size and position, the circle remains a complete graphical image in the graphical region, even though it might be clipped or otherwise altered on the display. Changing the shape or position of a region's viewport or window has no effect on the contents of the region itself, but only on what part of the region is visible on the display, where on the display it appears, and in what size and shape it is shown.

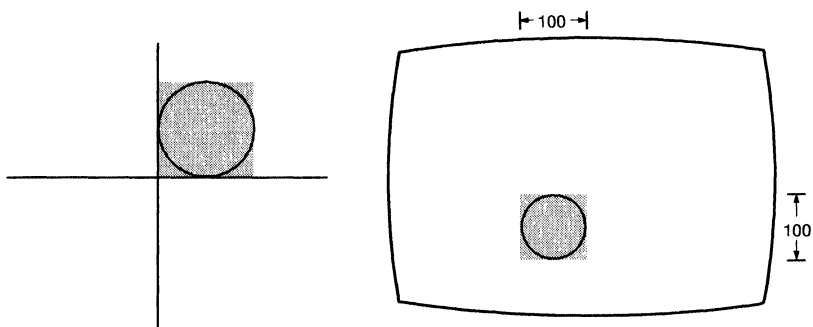
7.3.2 Changing the Viewport Size

You can change the size of a graphical region's viewport (or of all the viewports in a class) with the **change size** action statement. The coordinates are specified in positive or negative integer values in the coordinate system of the region's parent. Changing the viewport size of a graphical region does not affect the size of its window unless the **no scale** attribute has been specified. If **no scale** is specified, the window size changes in conjunction with the viewport size.

The **change size to** statement changes the absolute size of the windows by values representing the new X and Y coordinate positions of the graphical region's size. For example, suppose you change the size of Blackboard's viewport from its defined size of 150 150:

```
response to ChangeSize
  change Blackboard size to 100 100
```

As always, the entire contents of the window are shown in the viewport. The viewport has been made smaller, so the contents of the window appear smaller. You now see the following:

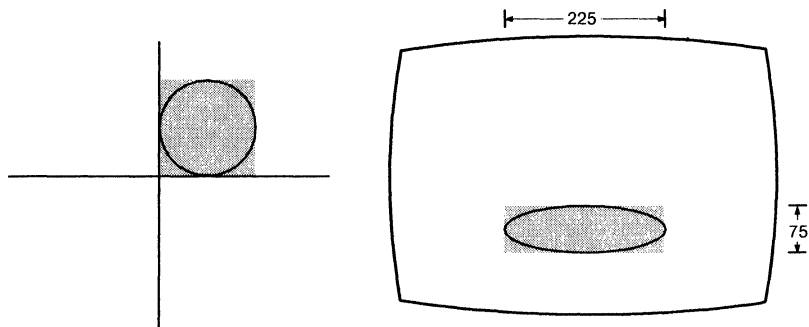


Note that the proportions of the viewport have not been changed. It is still a square, although each side is now 100 coordinate positions in length rather than 150. Therefore, the contents of the window (in this case, a circle) are displayed without distortion.

You can change the relative size of the viewport by using the **change size by** statement. For example:

```
response to ChangeSize
  change Blackboard size by 75 -75
```

makes the size of the viewport 75 coordinate positions wider in the horizontal dimension and 75 coordinate positions shorter in the vertical dimension. Its new size is 225 75. The viewport and its contents appear as shown below:



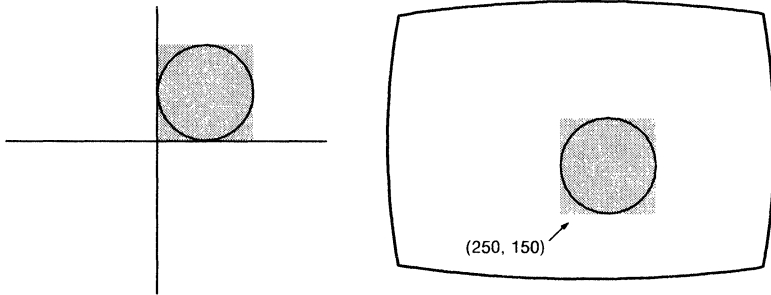
7.3.3 Changing the Viewport Position

You can change the position of a graphical region's viewport (or of all the viewports in a class) with the **change position** action statement.) The coordinates are specified as positive or negative integer values in the coordinate system of the region's parent.

The **change position by** statement changes the *relative* position of the graphical region's window. For example, suppose you move the viewport, from its defined position of 225 150 on the screen, to the right, as shown below:

```
response to ChangePos
  change Blackboard position by 25 0
```

You now see the following:

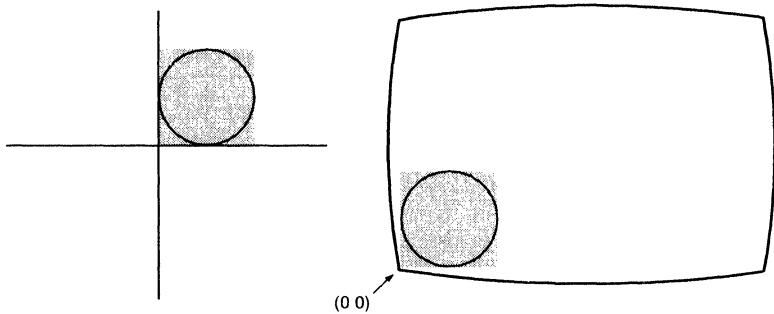


The contents of the viewport are exactly the same, because the window is unchanged and the size of the viewport is not affected. The viewport, however, appears farther right on the display. Its lower left corner is at coordinate position 250 150.

The **change position to** statement changes the *absolute* position of the viewport. For example:

```
response to ChangePos
  change Blackboard position to 0 0
```

moves the viewport to position 0 0 on the display screen. The viewport and its contents are shown below. Again, the contents of the viewport have not changed.



7.3.4 Changing the Window Size

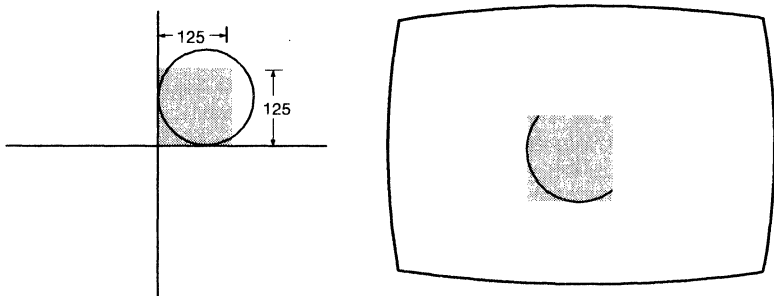
You can change the size of a graphical region's window (or of all the windows in a class) with the **change window size** action statement. The coordinates are specified in positive or negative integer values in the region's coordinate system.

The **change window size by** statement changes the *relative* size of the graphical region's window. For example, suppose you change the size of Blackboard's window from its defined size of 150 150, as shown below:

response to ChangeSize

change Blackboard window size by -25 -25

Less of the graphical region appears in the viewport because you have reduced the size of the graphical region's window. Note that changing the size of the window does not change the origin of the window.

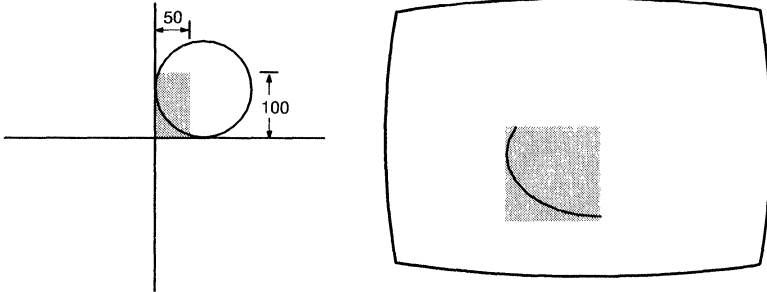


Only the part of the circle that is shown above in the rectangle of the viewport is now visible. Note that the part of the circle that is visible appears larger than it was before, because the contents of the window are magnified to fill the viewport completely.

The **change window size to** statement changes the *absolute* size of the window. For example:

```
response to ChangeSize  
  change Blackboard window size to 50 100
```

makes the window smaller in both the horizontal dimension and the vertical dimension. Less of the circle is visible in the viewport, as shown below:



Note that the proportions of the window have changed (it is no longer a square), so that the part of the circle that you see in the viewport appears to be flattened. There is more magnification in the X direction than in the Y direction, leading to an effect of stretching the circle to be an ellipse.

If **no scale** has been specified for the region, **change window size** has no effect.

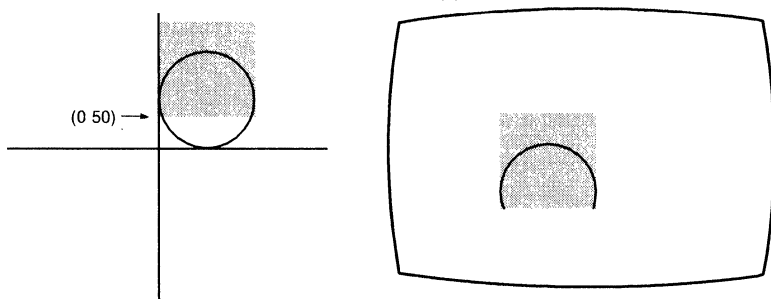
7.3.5 Changing the Window Position

You can change the position of a graphical region's window (or of all the windows in a class) with the **change window position** action statement. The coordinates are specified in positive or negative integer values in the graphical region's coordinate system.

The **change window position by** statement changes the *relative* position of the graphical region's window. For example, suppose you move Blackboard's window from its initialized position, as shown below:

```
response to ChangePos  
  change Blackboard window position by 0 50
```

The viewport remains the same, but the part of the graphical region displayed in the viewport changes. Changed from its original position of being centered within the viewport, Round now appears as:



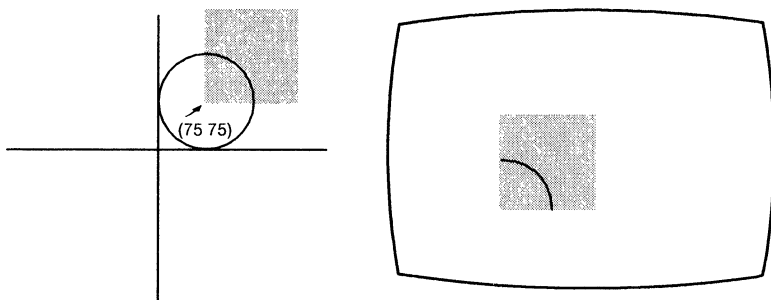
The window position has been moved up, so that what appears in the viewport is a higher portion of the graphical region. Only the top of the circle is now visible. Neither the size of the viewport nor the size of the window has been changed, so the part of the circle that is displayed has not changed size, nor has it been distorted in any way.

The **change window position to** statement changes the *absolute* position of the graphical region's window. For example:

response to ChangePos

change Blackboard window position to 75 75

The position of Blackboard's window has been moved from its initialized position of 0 0 in the graphical region. Blackboard's viewport displays only a quarter of the circle, as shown below. The remainder of the circle is no longer in the window, so it is clipped.



7.4 Using Built-in Functions with Graphical Objects

7.4.1 Introduction

The two most widely used ways in which object inquiry built-in functions are used during runtime are to base actions on the current attributes of an object, and to position objects.

Besides the object inquiry functions described below, you can also use the **xcoord** and **ycoord** response inquiry functions to inquire about an object's position. (These functions return the position of a selection of the current object.)

If you give an object inquiry function the name of an object that does not exist, an error message is produced.

Object inquiry functions do not return correct values when used in compile-time object definitions. For example:

Correct:

```
response to start
  add graphical region RegionA
    size (xsize of TR_1) (ysize of TR_1)
```

Incorrect:

```
graphical region RegionA
  size (xsize of TR_1) (ysize of TR_1)
```

7.4.2 Using Functions to Inquire about Size and Position

All of the functions related to size or position return integer values; thus, they can be used as operands in arithmetic expressions or can be compared to integer values in relational expressions.

Note that since a region's border is not part of a region's contents, it does not affect any of the functions that inquire about a region's size or position.

These functions work for textual regions exactly as they do for graphical objects, but the results that are returned are column and line numbers rather than X Y coordinates.

7.4.2.1 The top [of], bottom [of], left [of], and right [of] Functions

The **top [of]**, **bottom [of]**, **left [of]**, and **right [of]** functions can be applied to any object. The values returned are based upon the object's contents (drawing statements) and its children's contents. The values are expressed in the coordinates of the object. If the object has no contents, EASEL OS/2 EE returns values of zero for these functions.

Together, the values of these functions describe the smallest rectangle that encloses everything drawn in the object, including the object's children—even those that are invisible. If the object is a graphical region which has other graphical regions as children, the rectangle also encompasses the viewports of those children. The rectangle is not constrained by a graphical region's window size; the top, bottom, left, and/or right of the rectangle can lie either inside or outside the window.

CAUTION:

Avoid using the top [of], bottom [of], left [of], or right [of] functions for a scaled graphical region that contains image text, or in a program whose defined screen size is different from the display resolution. Because you cannot scale image text, top, bottom, left, and right may not work when used at compile time. You can use top, bottom, left, and right with image text at runtime.

For a graphical object, the following values are returned:

top [of]	The topmost Y coordinate occupied by the contents of the object.
bottom [of]	The bottom most Y coordinate occupied by the contents of the object.
left [of]	The leftmost point X coordinate occupied by the contents of the object.
right [of]	The rightmost X coordinate occupied by the contents of the object.

In the example below, the user wants to add pattern Picture to disabled key Logo. This pattern is added 10 positions to the right of the previously existing contents of Logo. The variable MaxLogo shows an estimated Maximum, beyond which no new contents will be added; if Logo is already to the right of that value, an error occurs.

```
if (right of Logo < MaxLogo) then    # Any more room?
  add to Logo
    move to (right of Logo + 10) Height
    pattern Picture
  else
    make ErrorMessage visible
end if
```

7.4.2.2 The `xmiddle [of]` and `ymiddle [of]` Functions

The **`xmiddle [of]`** and **`ymiddle [of]`** functions can be applied to any object. The values returned are the X and Y coordinates corresponding to the middle of the specified object's contents, and are based upon the values of the **`top [of]`**, **`bottom [of]`**, **`left [of]`**, and **`right [of]`** functions. The values returned are expressed in the coordinates of the object. If the object has no contents, EASEL OS/2 EE returns values of zero for these functions.

CAUTION:

Avoid using the `xmiddle [of]` or `ymiddle [of]` functions for a scaled graphical region that contains image text, or in a program whose defined screen size is different than the display resolution. Because you cannot scale image text, `xmiddle` and `ymiddle` may not work when used at compile time. You can use `xmiddle` and `ymiddle` with image text at runtime.

For a graphical object, the values are calculated as follows:

`xmiddle` = (`left` + `right`) / 2

`ymiddle` = (`top` + `bottom`) / 2

For example, if function values are:

`top` = 100

`bottom` = 50

`left` = 0

`right` = 100

then:

`xmiddle` = (0 + 100) / 2 = 50

`ymiddle` = (100 + 50) / 2 = 75

The following example draws a dot at the center of the disabled key Box:

```
response to MakeDot
  add to Box
    move to (xmiddle of Box) (ymiddle of Box)
    circle radius 1
```

7.4.2.3 The `xcursor [of]` and `ycursor [of]` Functions

The **`xcursor [of]`** and **`ycursor [of]`** functions can be applied to any object. The values returned are the coordinates of the graphics cursor in the specified object, and are expressed in the object's own coordinates.

The following example saves the cursor position into a pair of variables, so that the cursor can be put back to that position after the pattern is drawn.

```

response to AddDot
    copy (xcursor of Z) to SaveX
    copy (ycursor of Z) to SaveY
    add to Z
        pattern Dot
    move to SaveX SaveY

```

7.4.2.4 The **xposition [of]** and **yposition [of]** Functions

The **xposition [of]** and **yposition [of]** functions can be applied to any object. The values returned are the X and Y coordinates of the origin of the specified object, and are expressed in the coordinates of the object's parent.

In the following example, the user wants to move graphical region Zoom to the left by a given amount called StepSize. Before moving, the program checks to make sure that the position is such that Zoom can be moved without going off the edge of the screen.

```

response to MoveZoomLeft
    if (xposition of Zoom > StepSize) then
        change Zoom position by (-StepSize) 0
    end if

```

7.4.2.5 The **xsize [of]** and **ysize [of]** Functions

The **xsize [of]** and **ysize [of]** functions can be applied to graphical or textual regions only. When applied to a graphical region, the values returned are the X and Y dimensions of the specified graphical region's viewport.

In the following example, the user wants to move graphical region Chapter to the right by a given amount called StepSize. Before moving, the program checks to be sure that the position is such that Chapter can be moved without going off the right edge of the screen. To make this check, the program considers both the current position of Chapter (which marks its left edge) and its width. Note that (**xposition + xsize**) is the right edge.

```

response to MoveRight      # User selects right arrow
    if ((xposition of Chapter + xsize of Chapter)
        <= ScreenWidth - StepSize) then
        change Chapter position by StepSize 0
    end if

```

7.4.2.6 The window **xsize** [of] and window **ysize** [of] Functions

The **window xsize** [of] and **window ysize** [of] functions can be applied to graphical and textual regions only. For a graphical region, the values returned are the X and Y dimensions of the size of the graphical region's window, and are in the graphical region's own coordinates. If EASEL OS/2 EE returns values of zero for these functions, you may have inquired about a key.

In the following example, the user makes the window of Graph smaller in the Y dimension. After doing that, the program checks to see whether the new window height is large enough to be scaled down again. (The program will not allow the window height to go below zero.) If it is too small to be scaled down again, the program removes the scale-down request key from the screen so that it cannot be selected again.

```
response to StretchHeight
  change Graph window size by 0 (-Squish)
  if (window ysize of Graph <= Squish) then
    make StretchHeight invisible
  end if
```

7.4.2.7 The window **xposition** [of] and window **yposition** [of] Functions

The **window xposition** [of] and **window yposition** [of] functions can be applied to graphical and textual regions only. For a graphical region, the values returned are the X and Y coordinates of the origin of the graphical region's window, and are in the graphical region's own coordinates. If EASEL OS/2 EE returns values of zero for these functions, you may have inquired about a key.

In the following example, the user wants to pan the window of Book to the left, so that more of Book to the left can be seen. After doing this, the program checks to see if further requests to pan left are meaningful; if there are no more contents to the left, the program makes the key PanLeft invisible. Also, the program checks to see if requests to pan right are meaningful; if there are contents to the outside of the window, the program makes the key PanRight visible. Note that (**window position** + **window xsize**) is the right edge of the window.

```
response to PanLeft
  change Book window position by (-StepSize) 0
  if (window xposition of Book <= left of Book) then
    make PanLeft invisible
  end if
  if (window xposition of Book +
    window xsize of Book < right of Book) then
    make PanRight visible
  end if
```


7.4.3 Using Built-in Functions to Inquire about an Object's Attributes

7.4.3.1 The ancestry of Function

The **ancestry of** function can be applied to any object. The value returned is the string associated with the parent(s) of the specified object.

The following example illustrates the syntax used to represent ancestry. If the object Child1 is defined as:

```
Child1 in ParentN in ... in Parent2 in Parent1
```

the **ancestry of** function would return:

```
Parent1/Parent2/.../ParentN
```

For example, if you want to add a new object into the coordinate system of the parent of an existing object, you could use the **ancestry of** function:

response to AddWindow

```
add blue graphical region NewRegion size 512 280
  at position XCalculated YCalculated
  in ancestry of BigWindow
```

If you specify the **ancestry of** function for an object for which no ancestry has been defined, EASEL OS/2 EE returns the name of the primary region for windowed applications; "master" for fullscreen applications, indicating that the object was created in the screen—that is, it has no parent.

Be careful not to confuse the **ancestry of** function with the **ancestry** function. The **ancestry** function is a response inquiry, returning the string associated with the ancestry of the object that stimulated the response.

7.4.3.2 The background of and foreground of Functions

The **background of** function can be applied to regions only; the value returned is the integer value of the color associated with the background of the specified region.

The **foreground of** function can be applied to any object. When applied to regions, the value returned is the integer associated with the specified region's foreground color; when applied to keys, the value returned is the integer value associated with the specified key's color. For example, if you defined a graphical region as follows:

```
white graphical region Bulletin size 40 50
  at position 10 20
  red foreground
```

and then later specify in your program:

```
response to Mute
    make Labels color background of Bulletin
    make Labels foreground color foreground of Bulletin
```

EASEL OS/2 EE returns the integer value **7** in response to the **background of** Bulletin inquiry (making the background color of Labels white), and returns the integer value **1** in response to the **foreground of** Bulletin inquiry (making the foreground color of Labels red).

If you are inquiring about a textual region, you can use the **background at** and **foreground at** functions to find the background and foreground colors of a specified column and line in the textual region.

7.4.3.3 The border of Function

The **border of** function can be applied to a graphical or textual region. The value returned is the integer value of the color associated with the border of the specified object.

If the border is invisible, EASEL OS/2 EE returns a negative integer value of the border color. If no border is defined, EASEL OS/2 EE returns the value **-7** (invisible white). For example, if you define a graphical region as follows:

```
green graphical region Go size 100 50 at position 10 10
    blue border
```

then create a new object with the same border color:

```
response to NewScreen
    add green graphical region GoSecond size 20 20
    at position 10 10 in Screen2
    color (border of Go) border
```

EASEL OS/2 EE returns the value **4** (for the color blue). (See 5.3.3.1, "Standard EASEL OS/2 EE Color Set" on page 5-9.)

7.4.3.4 The exists Function

The **exists** function can be applied to any object. The value returned is the boolean (**true** or **false**) associated with the existence of the specified object. If the specified object currently exists (it has been defined, and has not been deleted), the value **true** is returned; otherwise, the value **false** is returned.

If you have more than one object with the same name, be sure to specify ancestry with the **exists** function. For example, if you define two graphical regions as follows:

```
graphical region News size 10 30
    at position 40 50 in Profits
    :
graphical region News size 50 50
    at position 15 15 in SalesWindow
    :
```

and then later specify in your program the **exists** function either as:

```
if (exists News in Profits) then ... # with ancestry
```

or as:

```
if (exists News) then ... # without ancestry
```

EASEL OS/2 EE returns the boolean value **true** in either case.

7.4.3.5 The handle of Function

The **handle of** function can be applied to any object except keys and sense regions. The function returns an integer value that is the PM handle (or identifier) for the object. This value can be passed to external functions and subroutines that can use the value.

Note that the consequences of the use of this value by an external function or subroutine are the responsibility of the developer. EASEL OS/2 EE does not guarantee proper operation when this handle is used by external code. (For instance, the code must not delete the window.)

7.4.3.6 The parameter of Function

The value returned for the **parameter of** function is the string value representing the parameter string associated with the specified object. For example:

```
cyan graphical region Delete size 80 25
    at position 50 10 in ActionBar
    parameter is "Deletepull"
```

```
response to line "to Delete" from Host
    make parameter of Delete visible
```

If you specify the **parameter of** function for an object for which no parameter has been defined, the value returned is the null string. (When an object is defined, the default value for the parameter attribute is the null string.)

Be careful not to confuse the **parameter of** function with the **parameter** function. The **parameter** function is a response inquiry, returning the parameter string associated with the object that stimulated the current response.

7.4.3.7 The priority of Function

The **priority of** function can be applied to any object. The value returned is the integer value of the priority associated with the specified object. For example, if you have defined a disabled key as follows:

```
red disabled key Warning1 at position 250 250
    priority is 3
```

and then later specify in your program:

```
priority of Warning1
```

EASEL OS/2 EE returns the value "3."

7.4.3.8 The selectability of Function

The **selectability of** function can be applied to any object. The value returned by EASEL OS/2 EE is the boolean (**true** or **false**) associated with the selectability of the specified object. If the object is selectable, the value **true** is returned; if it is not selectable, the value **false** is returned. For example, if you define a key:

```
key Hold at position 100 100
```

and then later specify in your program the function

```
selectability of Hold
```

EASEL OS/2 EE returns the value **true**.

7.4.3.9 The visibility of Function

The **visibility of** function can be applied to any object. The value returned by EASEL OS/2 EE is the boolean (**true** or **false**) associated with the visibility of the specified object. If the object is visible, the value **true** is returned; if it is invisible, the value **false** is returned. For example, the following code checks the visibility of an object to determine if the application is in the proper "state":

```
response to GoButton
    if (visibility of PleaseWait) then
        action MustWait
    else
        action GoAhead
    end if
```

7.5 Image Regions

Image regions allow you to display, scale, and scroll bitmap images created by Microsoft Paint™, PC PaintBrush®, and Presentation Manager's (PM) 1.1 standard bitmap format. EASEL OS/2 EE's graphical regions are vector-based graphics, in which images are created by describing them as line segments forming shapes on the screen. Bitmap images, by contrast, are created by describing the image as a grid of positions (pixels) each having a color value (for pseudo-color images) or being either on or off (for binary images).

Each imaging technique has advantages and disadvantages, involving trade-offs between speed and memory. Bitmap images easily provide a level of detail that is difficult to obtain with vector-based graphics and the images can be displayed significantly faster than comparable vector-based graphics. However, bitmap images can require vast amounts of memory, both on disk and in RAM.

Techniques developed to cope with this memory problem have in turn generated additional problems. For example, to reduce storage requirements on disk, the bitmap image files are usually encoded using some data compression scheme. Unfortunately for compatibility, there are numerous coding schemes, and thus bitmap file formats. Also, bitmap images are sometimes drawn piecemeal to reduce RAM requirements, and this reduces their speed. Support for bitmap images has been implemented with these considerations in mind.

7.5.1 Bitmap File Formats Supported

Initially, three file formats are supported: PM's standard version 1.1 bitmap, Microsoft Paint, and PC PaintBrush. You can use the PM Toolkit's icon editor or EASEL OS/2 EE (as described 7.5.9, "Read/Write Statements" on page 7-70) to create a PM bitmap format file. You can also add new file formats. The PM format can be either binary or pseudo-color; Microsoft Paint is binary only; and PC PaintBrush is binary only. A binary bitmap is displayed in two colors (black background with white foreground by default) and pseudo-color bitmaps are displayed in 16 colors. To cope with RAM requirements, you can control the amount of an image retained between displays. This allows you to trade speed and memory on an image-region by image-region basis.

Microsoft Paint is a trademark of Microsoft Corporation.

PC Paintbrush is a registered trademark of ZSoft Corporation.

7.5.2 Operations Supported

The normal region operations of windowing and scaling are also available for image regions. However, because scaling down a bitmap image entails losing information, you have some control over what information is lost. You can specify in the image region definition the scaling algorithm to be used in scaling down that image region (see 7.5.6, “Scaling” on page 7-67). In addition to the ordinary operations, you can specify color mapping for each image region. This allows you to control the appearance of an image region by changing color assignments, either when you define the image region or dynamically in response to some user input. The color mapping is specified separately with an image map specification, and can be used for more than one image region.

7.5.3 Defining Image Regions

The initial attributes of an image region are defined in an **image region** statement. You can change any of these attributes, except the scaling and cache specifications, during runtime. The definition uses the following model.

Model

```
[ enabled | disabled ] [ visible | invisible ] [ [color] COLOR]
[primary] image region IR_NAME
    size XSIZE YSIZE
    at { [position] X Y | default position }
    [in PARENT_NAME1 [in PARENT_NAME2]... ]
    [window size WINDOW_XSIZE WINDOW_YSIZE
      at [position] WINDOW_X WINDOW_Y]
    [ [color] COLOR foreground]
    [priority is PRIORITY_NO]
    [parameter is PAR_STRING]
    [ no scale | sample | preserve foreground | preserve background ]
    [ no cache | cache image | cache viewport | cache CACHE_SIZE bytes ]

[REGION_ATTRIBUTES]

[file FILE_NAME [using IMAGE_MAP] ]
```

The REGION_ATTRIBUTES that you can use to define the appearance of an image region are presented in 10.1, “Region Attributes” on page 10-2.

Only the **region** specification, the identifier, the **at** [**position**] specification, and the **size** specifications are required. Other attributes are optional, using the defaults shown below.

SELECTABILITY:	enabled
VISIBILITY:	visible
COLOR (background):	black
ANCESTRY:	in primary region (in master for fullscreen)
COLOR (foreground):	white
PRIORITY:	0
PARAMETER:	""
BORDER:	invisible
COLOR (border):	white

Except for the **file** option, which must be specified last, any of the attributes after ancestry can be specified in any order.

7.5.4 Foreground and Background Colors

The background and foreground colors are used in binary bitmap images to specify the two colors to display the image in, but only the background color has significance in pseudo-color images. As with other region types, the user can change these colors at runtime.

7.5.5 Cache Specifications

The cache is the part of RAM used to hold the portion of the bitmap image that persists between display requests. The data in the cache is uncompressed, scaled, and color mapped, and thus is immediately ready for display.

The cache specifications allow you to control how much memory EASEL OS/2 EE uses for image regions, from potentially a lot in the case of a cache image, to as little as desired in the case of a work area cache. You can choose from four cache specifications: **cache image**, **cache viewport**, work area cache (**cache N bytes**), and **no cache**. The other three use program memory; **no cache** uses system memory.

7.5.5.1 Cache Image

A cache image contains the entire image from the bitmap file, scaled according to the window to viewport translation. It normally takes several seconds for the first display of an image while EASEL OS/2 EE gets the entire image into memory, but subsequent displays (until you resize the image region) are nearly instantaneous. An image region can have initial contents that are read in at compile time (see 7.5.7, "Specifying File Contents" on page 7-68), in which case even the first display is nearly instantaneous. The disadvantage of a cache image is that it requires large amounts of memory, and due to internal memory allocation limitations the

user is actually restricted to images of less than 64K. Therefore using **cache image** is not recommended if the image will be resized to be larger. This is the default cache specification.

7.5.5.2 Cache Viewport

A cache viewport contains just the part of the image that is visible on the screen. It usually comes up faster initially, because less overhead is involved getting only the visible portion of an image into memory. Like a cache image, subsequent displays are nearly instantaneous. However, if the image is bigger than the viewport, scrolling is slowed because new portions must first be decoded, color mapped, and scaled. A cache viewport is handy for an image region that is made invisible and visible again, because that operation is nearly instantaneous.

7.5.5.3 Work Area Cache

A work area cache is an arbitrarily sized area of memory (although it must be big enough to hold at least one scanline) that is used as a temporary location to build the image as it is being displayed. It is intended to be used when the user wants to control the exact amount of memory used because memory is scarce, or if the image will be resized to be larger. Usually this amount is much smaller than a typical viewport and thus the image appears a piece at a time. The more pieces the image is broken down into, the more overhead incurred and consequently the more time taken to display the image. The larger the work area cache, the fewer pieces the image must be broken down into. Only occasionally does a work area cache retain usable data across display requests. It comes in handy if an image region is partially obscured by another object that becomes visible and invisible repeatedly.

7.5.5.4 no cache

no cache, unlike the other cache types, does not refer to the size of the memory area, but to the type of memory used. The other three cache specifications tell EASEL OS/2 EE to use program memory, a portion of the memory allocated to EASEL OS/2 EE itself with the **-m** switch. The **no cache** specification tells EASEL OS/2 EE to use system memory, the memory that is left over and is normally used for local applications and other programs the system may be running.

no cache is not part of the EASEL OS/2 EE program, is not saved in the binary file, and thus must be created and filled each time the program is run. Also the cache is not permanently allocated so the system may discard the contents of the cache whenever system memory is needed. EASEL OS/2 EE determines the cache's size, based on how much memory is available when the image is displayed.

7.5.6 Scaling

Image regions lose information when they are scaled down. Suppose a line drawing of a house is represented as a binary bitmap image. Scaling down vertically by a factor of 3 means for every three scanlines in the original image the resulting image will have only one. You have three options for specifying what the resulting picture will look like: sample, preserve foreground/background, and no scale.

7.5.6.1 sample

The simplest and fastest scaling method is to arbitrarily retain one of every three scanlines, say the first, and discard the others. This method is called sampling, and is specified by the keyword **sample** in the image region definition. It is easy to imagine the picture seriously deteriorating if a horizontal line representing a floor in the house is one of the scanlines discarded. (If, instead, the image is a scanned representation of a picture of a house, the image is not likely to degrade significantly and the performance gain will be the stronger factor in choosing the scaling option.) **sample** is the default if you do not explicitly choose a scaling option and is the only method applicable to pseudo-colored images.

7.5.6.2 Preserve Foreground/Background

A second scaling method is to divide the image into groups of three scanlines. For each group one white output scanline is generated. Each scanline of the group is then taken in turn and for each horizontal position in that scanline that is black, black is put in the corresponding position in the output scanline. This ensures that lines are preserved in the reduced image. However, this method takes much longer, because each position of each scanline is inspected to produce the final image.

When the original line drawing is like a blueprint with white lines on a black background, you would want to preserve the white (the foreground) in each scanline. You would use **preserve foreground** as the scaling option in this case. Also, the image may not be black and white, but green and red, or any other color combination.

When the image region is defined, foreground and background colors are established, either by specifying them or by default. You can then specify either preserve foreground or preserve background to indicate which color EASEL OS/2 EE should preserve. So if the line drawing of a house is rendered white on blue in the original, with the foreground white and the background blue, simply specify **preserve foreground** in the image region definition.

7.5.6.3 No Scale

You can also specify the **no scale** option, which guarantees that the window size and the viewport size are always identical. When **no scale** is specified and you change window size, EASEL OS/2 EE ignores the change. When **no scale** is specified and you change viewport size, EASEL OS/2 EE changes both the viewport size and the window size. This ensures that no scaling is ever necessary, greatly reducing display time.

7.5.7 Specifying File Contents

You can specify initial contents for an image region in the same way you include files in textual regions. Although image regions never contain drawing statements, you can use the **file** option to specify a file that will be the image region's initial contents. EASEL OS/2 EE recognizes the file format (PM's standard bitmap, Microsoft Paint, or PC PaintBrush) from information in the header. EASEL OS/2 EE also determines whether the files are binary or pseudo-color, and the size of the image.

The file option must be specified last in the region definition. The file specification is simply the keyword **file** followed by a string value, either a literal or a string constant or a string with an initial value. If you do not specify an extension, EASEL OS/2 EE assumes the extension ".bmp". If you do not specify an extension, and there is no file with the extension ".bmp", EASEL OS/2 EE assumes that the file has no extension.

Specifying initial file contents for an image region does not ensure that the cache is filled before the first display. If you specify an image or cache viewport, the cache is filled before a display request is received, but if you specify a work area cache or no cache, no data is put in the cache until the cache is actually displayed at runtime.

7.5.7.1 Specifying an Image Map

An image-map specification allows you to map the colors of an image region by listing color number/color value pairs:

Model

```
imagemap IMAGEMAP_NAME is
  OLD_COLOR NEW_COLOR[, OLD_COLOR NEW_COLOR]...
```

where OLD_COLOR and NEW_COLOR are EASEL OS/2 EE color numbers. The actual values may be integer variables, integer constants, integer literals, or the EASEL OS/2 EE color keywords. You cannot use the format: color COLOR_NUMBER

The image map tells EASEL OS/2 EE how to color an image. Different software produces different color schemes so that a red object in the original image might be displayed as blue in EASEL OS/2 EE. EASEL OS/2 EE supports 16 colors, and because some software can support more, undesirable results may occur as different colors in the original are mapped to the same color in EASEL OS/2 EE.

7.5.8 Specifying Image Maps in Image Regions

Use the **using** option, either in the image region definition with the file option, or in a read action statement when you want to associate an image map with an image region. The syntax is as follows:

```
file FILE_NAME using IMAGEMAP_NAME
or
read file FILE_NAME into IMAGE_REGION_NAME using
    IMAGEMAP_NAME
as in this example: string CurrentFile is "file1.pcx"

integer variable Black is 0
integer variable Red is 1
integer variable Green is 2
integer variable Yellow is 3
integer constant Blue is 4
integer constant Magenta is 5
integer constant Cyan is 6
integer constant White is 7

imagemap Colormap1 is      # you can use:
    black red,             # two keywords
    White Magenta,        # two constants
    Red Yellow,           # two variables
    4 1                    # two literals

imagemap Colormap2 is      # you can use:
    red Magenta,          # a keyword and a constant
    white Black,          # a keyword and a variable
    green 7,              # a keyword and a literal
    Blue 2,               # a constant and a literal
    Cyan Yellow,          # a constant and a variable
    Black 5,              # or a variable and a literal

image region One
```

```
size 640 350 at 0 0
preserve foreground
cache image
file CurrentFile using Colormap1
```

```
response to char "1" from keyboard
  read file CurrentFile into One using Colormap1
```

```
response to char "2" from keyboard
  read file "file2.pcx" into One using Colormap2
```

In this example, when Colormap1 is used, everything that was black in the original image becomes red; everything that was white becomes magenta; everything that was red becomes yellow; and everything that was blue becomes red.

Likewise, when Colormap2 is used, everything that was red in the original image becomes magenta; everything that was white becomes black; everything that was green becomes white; everything that was blue becomes green; everything that was cyan becomes yellow; and everything that was black becomes magenta.

7.5.9 Read/Write Statements

The **read** and **write** action statements that apply to textual regions apply to image regions. The read statement allows the user to fill an image region's contents dynamically at runtime, perhaps in response to some event. Unlike a textual region, a file statement specified for an image region can have a **using** clause to indicate the desired color mapping:

```
read file "skier" into Display2 using Colormap2
```

The using clause is optional, and you do not specify one if you do not want color mapping. If you do not specify an extension, EASEL OS/2 EE assumes the extension ".bmp". If you do not specify an extension, and there is no file with the extension ".bmp", EASEL OS/2 EE assumes that the file has no extension. EASEL OS/2 EE does not distinguish between colored and uncolored files when you read files into image regions (unlike textual regions) so while the keyword **colored** does not generate an error, it makes no sense in this instance.

The write statement allows the user to create an EASEL OS/2 EE format file containing the visible contents of image and graphical regions. The write is always destructive, replacing the contents of any existing file. Specifying the append option has no effect because it is ignored unless a textual region is being written. The syntax is similar to that for a textual region write statement:

```
write REGION_NAME to file FILE_NAME
```

As with the read statement, EASEL OS/2 EE does not distinguish between colored and uncolored files when you write files to image regions, so the keyword **colored** makes no sense. The file is in PM standard bitmap format and is a pseudo-color file.

When a write statement is executed, EASEL OS/2 EE copies the portion of the screen where the region is or would be displayed. Therefore, even if the region is currently invisible or obscured by other objects, the portion of the screen where the region would be displayed, or that is directly above that region, is captured on disk.

If the region extends beyond the screen, it is clipped to the screen for the output file. This is true of an image region as well, whether it uses a cache image or a work area cache, and if it is color mapped, the current color map is used. Essentially a snapshot is taken of the part of the screen where the specified viewport intersects it.

7.6 Sense Regions

Sense regions allow you to define a rectangular portion of a graphical or image region that can be selected. For instance, if you were displaying a wiring diagram and three parts on the diagram could be selected for more information, you could use three sense regions to define those parts of the diagram that are selectable.

Sense regions can overlap other objects and can be used to enable or disable a large area of the screen. Sense regions themselves have no graphical contents or visual appearance; they cannot display frame style attributes, and they cannot have children.

7.6.1 Defining Sense Regions

The initial attributes of an sense region are defined in an **sense region** statement. You can change any of these attributes during runtime. The definition uses the following model.

Model

```
[ enabled | disabled ] [ visible | invisible ]
sense region SR_NAME
  size XSIZE YSIZE
  at { [ position ] X Y | default position }
  [ in PARENT_NAME1 [ in PARENT_NAME2 ] ... ]
  [ window size WINDOW_XSIZE WINDOW_YSIZE
    at [ position ] WINDOW_X WINDOW_Y ]
  [ priority is PRIORITY_NO ]
  [ parameter is PAR_STRING ]
```

Only the **sense region** specification, the identifier, the **at** [**position**] specification, and the **size** specification are required. Other attributes are optional, using the defaults shown as follows.

SELECTABILITY:	disabled
VISIBILITY:	visible
ANCESTRY:	in primary region (in master for fullscreen)
WINDOW_XSIZE:	XSIZE
WINDOW_YSIZE:	YSIZE
WINDOW_X:	0
WINDOW_Y:	0
PRIORITY:	0
PARAMETER:	""

7.6.1.1 Visibility Attribute

Although sense regions have no visual appearance on the screen, the visibility attribute of a sense region does determine whether it has an effect on user selections. For example:

If a sense region is: **The user:**

invisible	cannot select it
enabled and visible	can select it
disabled and visible	cannot select it or objects underneath it

7.6.1.2 Priority Attribute

You can use the priority attribute to determine whether the sense region is above or below its siblings. (The priority value can be any integer, positive or negative. Objects with higher priority are always on top of objects with lower priority.) While this ordering has no visual effect, it does determine how objects are selected. For example, if two objects overlap at the point of a user selection, the higher of the two objects is selected. Placing a visible disabled sense region over other enabled objects prevents the user from selecting those objects.

7.6.2 The **xcoord** and **ycoord** Built-in Functions

The window size and position of a sense region determine how a user selection point is converted into the built-in functions **xcoord** and **ycoord**, in the same manner as for graphical and image regions. This feature is primarily useful if the sense region is covering another region whose window size or position is also changing.

7.6.3 PM Frame Attributes

Sense regions cannot be used over PM frame attributes (title bars, size borders, etc.).

Chapter 8. Textual Regions

8.1 Defining and Using Textual Regions

8.1.1 Overview

Textual regions are similar to graphical regions in many ways. Like graphical regions, they have attributes such as visibility, selectability, border, and color; they are logically limitless in size and have viewports and windows; and their positions are specified in X Y coordinates. But a textual region is, as its name implies, solely for the purpose of displaying and manipulating large amounts of text. A textual region lets you, for example, compose and edit a memo, present a textual legend alongside a graphical display, or display a table of textual data.

A textual region cannot contain graphics. Because of this, there are many ways in which a textual region differs from a graphical region; namely:

Its coordinate system, which is based on columns and lines rather than X Y coordinates. (There are no zero or negative lines or columns.) The topmost line in a textual region is line 1, and line numbers increase as you move downward. For example:

move to column 8 line 3

tells EASEL OS/2 EE to move to the eighth character position in the third line of the textual region.

Its window, which is in the coordinate system of the textual region. Thus, a textual region's window is defined in terms of the number of columns of text that fit horizontally in the window, and the number of lines of text that fit vertically in the window.

Its cursor. A textual region has only a text cursor; it does not have a graphics cursor. A textual region drawing statement begins at the current position of the text cursor, just as a drawing statement for a graphical region begins at the current position of the object's graphics cursor. When you move the text cursor in a textual region, you specify the column and line numbers of the text, rather than X Y coordinates (which are used for graphical objects).

Although the text cursor is normally not visible on the display, you can make the text cursor visible in a textual region by specifying a **make cursor visible** statement.

Its ancestry. A textual region cannot be the parent of any other object.

Its drawing statements, which are distinct from the set of graphical drawing statements. The drawing statements for textual regions manipulate textual data only; for example:

```
move to column 10 line 16      # positions text cursor
insert "hello"                 # display text
```

If you use a graphical drawing statement in the contents of a textual region, or if you use a textual region drawing statement in the contents of a graphical object, EASEL OS/2 EE will report an error condition.

Its fonts. Only one font can be used in a textual region at any given time. You cannot change this font during runtime. You can use only fixed-width image fonts in textual regions; you cannot use proportional or outline fonts. If you specify a proportional or an outline font in a textual region, it is replaced by the default font. If the default font is proportional or scalable, font **medium** is used.

Its extra action statements. Many action statements used for keys and graphical regions are also used for textual regions, but textual regions have many extra action statements that manipulate the text in various ways.

8.1.2 Defining a Textual Region

The initial attributes of a textual region are defined in a **textual region** statement. You can change any of these attributes, except the font, during runtime. The definition uses the following model.

Model

```
[ enabled | disabled ] [ visible | invisible ] [ [color] COLOR ]
[primary] [colored] textual region TR_NAME
{ size XSIZE YSIZE | window size COLUMNS columns LINES lines }
at { [position] X Y | default position }
[in PARENT_NAME1 [in PARENT_NAME2]... ]
[ [color] COLOR foreground ]
[priority is PRIORITY_NO]
[parameter is PAR_STRING]
[font FONT_NAME]

[REGION_ATTRIBUTES]

[DRAWING_STATEMENT]...
```

The REGION_ATTRIBUTES that you can use to define the appearance of a textual region are presented in 10.1, "Region Attributes" on page 10-2.

Any of the attributes after ancestry can be specified in any order.

Only the **textual region** specification, the identifier, the **at [position]** specification, and the **size** or **window size** specifications are required. Other attributes are optional, using the defaults shown as follows.

SELECTABILITY:	enabled
VISIBILITY:	visible
COLOR (background):	black
ANCESTRY:	in primary region (in master for fullscreen)
COLOR (foreground):	white
PRIORITY:	0
PARAMETER:	" "
BORDER:	invisible
COLOR (border):	white
FONT:	medium¹

8.1.2.1 Defining the Size of the Window and Viewport

The size of the textual region's viewport is defined in X Y coordinates with the **size** specification, or is defined in lines and columns with the **window size** specification. You cannot define both. For textual regions, the size of both the viewport *and* the window is determined by using one of these specifications.

The viewport of a textual region lies in the coordinate system of the parent of the textual region. Since a textual region cannot be a parent of any object, the parent of the textual region must be a graphical object, and thus the viewport size must be specified in X Y coordinates. For example, the following statement defines a textual region's size, which is also the viewport's size, in X Y coordinates:

```
textual region WriterHere size 50 200 at position 0 50
```

With the **size** specification, the size of the window is inferred. That is, the number of columns in the window is set to the largest number that a viewport of this size can display horizontally, and the number of lines in the window is set to the largest number that a viewport of this size can display vertically, based upon the textual region's font.

If you use the **window size** specification to define the size of the textual region, its size must be specified in columns and lines, since the window lies in the coordinate system of the textual region. For example, the following statement defines the size in columns and lines:

```
textual region News window size 60 columns 35 lines  
    at position 0 50
```

¹ This is the font used if none is specified as an environment declaration.

With the **window size** specification, the size of the viewport is made to be the height and width necessary to display the complete window in the textual region's font.

When you use **window size**, the actual space that the textual region takes up on the screen (in X Y coordinates) is based on the specified number of columns and lines and the size of the font. For example, in any given viewport you can fit more columns and lines of text displayed in EASEL OS/2 EE's "medium" font than in EASEL OS/2 EE's "large" font. If you use **window size** to define the size of the textual region, it will not matter what font you use—the displayed area will always be the number of columns and lines specified. Thus, it is recommended that you define all textual regions with **window size**, unless the textual region must occupy a specific area proportional to other objects on the monitor.

8.1.2.2 Defining and Using a Colored Textual Region

You can define a textual region as **colored**. In a colored textual region, you can specify background and foreground colors for individual characters. For example, you can display a textual region with mostly white text on a blue background, but also with some red text on a black background, some yellow text on a red background, etc. (The color attributes are preserved when you save a program see 11.3.5, "Saving a Program in Progress" on page 11-12.)

The color specifications in the definition are for the background and foreground of the entire region, exactly like an uncolored textual region. The color specifications for *individual characters* in the region override the color specifications in the definition. You can specify the position and color for characters in two ways:

1. For text that already exists in a colored textual region in your program, you use the **make segment** or **make block** action statement to specify a character's position or color.
2. For text in an ASCII text file that will be read into the colored textual region, you add a "color header" to the file, and then use the **CText** Module (see 8.7.3, "Specifying Colors for Text in a File: Using the CText Module" on page 8-57). The **CText** module reads the information in the color header and applies the specified colors to the individual characters through EASEL OS/2 EE commands in the module. Both the color header and the text in the ASCII text file can be edited at any time and read into the colored textual region without any changes to the program.

When you assign a color to a character in a colored textual region, the color stays assigned to that character even if the character changes position later on—in other words, the specified color attributes are assigned to the characters, not to the character positions. For example, if text is inserted or deleted during runtime, the displaced characters will retain their colors as originally specified.

All of the drawing and action statements for uncolored textual regions also apply to colored textual regions. For example, you can read and write colored textual regions to and from files. When you write a colored textual region to a file, you can create a regular ASCII text file or a "*color-attributed*" file. A color-attributed file is a binary file that assigns color to the text on a character-by-character basis, similar to the way that a word processing program assigns attributes to its text files. Color-attributed files cannot be edited.

Special considerations for colored textual regions are noted in the descriptions for each of the textual region statements in this chapter. Whenever the words "textual region" are used without being distinguished from colored textual regions, the information applies to both types of textual regions.

8.1.2.3 Fonts in Textual Regions

The font specified in a textual region definition overrides the default font for the program. You can use only one font for an entire textual region; you cannot change this font during runtime. You can use only fixed-width image fonts in textual regions; you cannot use proportional or outline fonts. If you specify a proportional or an outline font in a textual region, it is replaced by the default font. If the default font is proportional or scalable, font **medium** is used.

You can increase or decrease the amount of displayed text by changing the size of the textual region's window and viewport (see 8.4.9, "Changing the Size and Position of a Textual Region's Window and Viewport" on page 8-36).

8.2 Drawing Statements for Textual Regions

8.2.1 Overview

Except for the **move** statement, none of the drawing statements used for graphical objects can be used for textual regions, because they use the X Y coordinate system. The drawing statements described in this section can be used only to modify a textual region. They move the text cursor, insert or overwrite text, select text segments and blocks, and otherwise manipulate text when you compile the program. If you specify one of these statements in the contents of a graphical object, the drawing statement will not be executed and an error message will be generated.

8.2.1.1 Where to Place Textual Region Drawing Statements

All textual region drawing statements are valid in object definitions and action statements. In object definitions, the initial appearance of a textual region is defined by the drawing statements that are specified in the contents of its definition. For example:

```
visible textual region Report size 300 50 at position 50 100
  insert "This is the text"
```

You can include drawing statements within action statements (which are themselves within response definitions) to modify the textual region. For example:

```
response to NewWord
  add to Report
    move to column 1 line 1
    overwrite "That"
```

Here, the **overwrite** drawing statement is included in the **add to** action statement, which is itself a part of the response definition to **NewWord**. Thus, when **NewWord** is selected, the word "This" is replaced with the word "That" in the **Report** textual region.

8.2.1.2 Addressing a Character

A character position in a textual region is addressed in terms of columns and lines, not in X Y coordinates. You can specify an absolute character position, or a position relative to the current position of the text cursor.

In a textual region, the positive directions are right and down (the same direction in which English text is written). The topmost line in a textual region is line 1, and line numbers increase as you move downward. You can specify movement in a negative direction; for example, the action statement below moves the text cursor 10 columns to the left and 5 lines up.

```
move -10 columns -5 lines
```

The column number and line number of a textual region refer to the entire textual region, not just to the part of the textual region that is in the window. For example, the window of a particular textual region might show only columns 5 through 45 of lines 13 through 18.

A textual region has only positive lines and columns; you cannot address a non-positive (0 or below) line or column number. (You can change the window position to include negative columns or lines, but this is not recommended; these columns and lines cannot contain any text, and will always be displayed in the background color of the textual region.)

If you define a textual region without specifying any contents for it, it will contain no columns or lines.

8.2.2 Moving the Text Cursor

In textual regions, the **move** statement works very similarly to how it works for graphical objects. The only exceptions are that (a) there is no graphics cursor in a textual region and (b) the text cursor is moved by specifying columns and lines, rather than by X Y coordinates. You can move the cursor to any character position, whether or not any text exists at that position.

The **move [by]** statement moves the text cursor to a relative position, by the number of columns and lines specified. For example, if the text cursor is at column 9, line 20 and you specify:

```
move 10 columns 5 lines
```

EASEL OS/2 EE moves the text cursor 10 columns to the right of and 5 lines down from its starting position, to column 19, line 25. (Remember, the line numbers increase as you move downward in a textual region.) If you specify negative positions; for example:

```
move -7 columns 5 lines
```

the text cursor moves 7 columns to the left of and 5 lines down from its starting position. Be careful when specifying relative movements: if they do not result in a positive coordinate (that is, column 1 line 1 or higher), the cursor will not move and an error message will be produced.

The **move to** statement moves the text cursor to a specified absolute position. For example, if you specify:

```
move to column 2 line 51
```

the text cursor moves to column 2, line 51, regardless of its starting position.

Absolute column and line numbers must be positive integers. An absolute negative or zero column or line specification will generate an error message, and the text cursor will not move.

8.2.3 Defining Text Segments and Text Blocks

8.2.3.1 Defining a Text Segment

A text segment is any set of contiguous characters within a textual region. You can copy a segment, use it to overwrite existing text, emphasize it, or remove it. To define a segment, use the following model. The segment definition itself must appear within an **insert** or **overwrite** drawing statement, or in an **emphasize** or **remove** action statement.

Model

```
segment [column] COL_NO [line] LINE_NO  
  [thru [column] COL_NO [line] LINE_NO]
```

In the definition, you specify integer values for the starting and ending column and line character positions for the segment. The segment includes the first and last character positions specified.

A segment can be defined by any two endpoints of the text (e.g., first character through last, or last character through first); for example, if you specify:

```
response to ...  
  add to NewTxt  
    insert segment column 3 line 1 thru column 8 line 3  
      from OldTxt
```

or:

```
response to ...  
  add to NewTxt  
    insert segment column 8 line 3 thru column 3 line 1  
      from OldTxt
```

the following text from the textual region OldTxt will be inserted into the textual region NewTxt:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	a	b	c	d	e	f	g	h	i	j	k		
2	l	m	n	o	p	q	r	s	t	u	v	w	x
3	y	z	a	b	c	d	e	f	g	h	i		
4													
5													
6													
7													
8													

If you omit the **column** and **line** keywords, the first and third integer values always specify the column numbers, and the second and fourth integer values always specify the line numbers. For example:

```
response to ...  
  add to NewTxt  
    insert segment 3 1 thru 8 3 from OldTxt
```

Again, beginning and ending positions can be reversed. The above statement is equivalent to :

```
response to ...  
  add to NewTxt  
    insert segment 8 3 thru 3 1 from OldTxt
```

You can define a segment that contains character positions to the right of the last character in a line, as long as that line is not the last one in the textual region. For example, if you specify:

```
response to ...  
  add to NewTxt  
    insert segment column 10 line 3  
      thru column 10 line 5 from OldTxt
```

and the textual region OldTxt contains the following text:²

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	a	b	c	d	e	f	g	h	i				
2	j	k	l										
3	m	n	o	p	q	r	s	t					
4	u	v	w	x									
5	a	b	c	d	e								
6													
7													
8													

the segment will begin at column 9, line 3 (the position following the last character in line 3), and end at column 6, line 5 (the position following the last character in line 5).

² Note that in these illustrations, the gray shaded box indicates the position of the text cursor.

However, you cannot define a segment that contains character positions that begin and end anywhere after the last character in the last line of the textual region. If you attempt to do so, no segment will be selected, and an error message will be produced. For example, if you had specified:

```
response to ...
  add to NewTxt
    insert segment column 10 line 5
      thru column 15 line 5 from OldTxt
```

an error message would have been generated, since the segment is after the last character in the last line of the textual region.

If **thru** is not specified—that is, if you provide only one set of column and line numbers—the segment will be a single character position.

8.2.3.2 Defining a Text Block

A block is any rectangular area within a textual region. You can copy a block, use it to overwrite existing text, replace it with blanks, or remove it. To define a block, use the following model. The **block** definition itself must appear within an **insert**, **overwrite**, or **blank** drawing statement, or in a **remove** action statement. (These statements are described later in this chapter.)

Model

```
block [column] COL_NO [line] LINE_NO
  [thru [column] COL_NO [line] LINE_NO]
```

In the definition, you specify integer values for the starting and ending column and line character positions for the block. The block includes the first and last character positions specified.

If you omit the **column** and **line** keywords, the first and third integer values always specify the column numbers, and the second and fourth integer values always specify the line numbers. If you omit the **thru** keyword—that is, if you only provide one set of column and line numbers—the block will be a single character position.

But here the similarities between blocks and segments end. Because a block defines a rectangular portion of a textual region, the block might not contain all contiguous characters from the beginning of the block to the end of it. For example, if you specify:

```
response to ...
  add to NewTxt
    insert block column 3 line 1 thru column 8 line 3
      from OldTxt
```

the following text block from the OldTxt will be inserted into NewTxt:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	a	b	c	d	e	f	g	h	i	j	k		
2	l	m	n	o	p	q	r	s	t	u	v	w	x
3	y	z	a	b	c	d	e	f	g	h	i		
4													
5													
6													
7													
8													

The block can be defined by any two opposite corners (e.g., upper left/lower right, upper right/lower left, lower left/upper right, or lower right/upper left). The following are all valid **block** definitions, which define the same text block as shown in the above illustration.

```
# upper left through lower right:
insert block column 3 line 1 thru column 8 line 3 from A
# or:
insert block 3 1 thru 8 3 from A

# lower right through upper left:
insert block column 8 line 3 thru column 3 line 1 from A
# or:
insert block 8 3 thru 3 1 from A

# upper right through lower left:
insert block column 8 line 1 thru column 3 line 3 from A
# or:
insert block 8 1 thru 3 3 from A

# lower left through upper right:
insert block column 3 line 3 thru column 8 line 1 from A
# or:
insert block 3 3 thru 8 1 from A
```

Also, unlike a segment, a block can begin and/or end anywhere beyond the last character of the last line in a textual region.

8.2.4 Inserting Text

The **insert** drawing statement, specified according to the model below, lets you add text to a textual region from several different sources. You can insert a string value, a specified segment or block of text from another textual region, an ASCII text file, or a color-attributed file. Because **insert** is a drawing statement, these statements can be executed at compile-time or at runtime.

Model

```
insert { [ at cursor | after cursor ] TEXT_STRING |
         [ at cursor | after cursor ]
         segment [column] COL [line] LN
               [thru [column] COL [line] LN] from TR_NAME |
         block [column] COL [line] LN
               [thru [column] COL [line] LN] from TR_NAME |
         [colored] file FILE_NAME }
```

You can move the cursor to any character position, whether or not any text is there. If text is added to a textual region at a location where there is no text, EASEL OS/2 EE will insert the necessary blanks prior to the text cursor. For example, if the textual region is empty, and you specify:

```
add to Greeting
  move to column 5 line 5
  insert "Hello"
```

When this statement is executed, the text cursor moves to column 5 line 5, and new lines and blanks are automatically inserted as shown below:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1													
2													
3													
4													
5		^	^	^	^	H	e	l	l	o			
6													
7													
8													

In these examples, padded blanks are shown by a caret (^).

8.2.4.1 Inserting a Text String

When an **insert** STRING statement is executed, the text is inserted by default beginning at the current position of the text cursor, and all text after the inserted text is moved to the right to make room for the new text. You can optionally specify **after cursor** to indicate that the text should be placed after the current cursor position. For example, if the text cursor is at column 3, line 3:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	T	h	i	s		i	s		a	n			
2	e	x	a	m	p	l	e		o	f			
3	a		d	o	c	u	m	e	n	t			
4													
5													
6													
7													
8													

and you specify:
add to Document
 insert "secret "

EASEL OS/2 EE inserts the text beginning at column 3 of line 3. The textual region after the insertion is shown below:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	T	h	i	s		i	s		a	n			
2	e	x	a	m	p	l	e		o	f			
3	a		s	e	c	r	e	t		d	o	c	u
4													
5													
6													
7													
8													

As shown in the illustration, the text cursor is now positioned on the character following the last inserted character. This is the same character on which the text cursor was positioned before the insertion, but it is now in a new column position. (Note here that the word “document” has not been truncated; it simply extends beyond column 13. To display the rest of the word, you just need to enlarge the window.)

8.2.4.2 Using Escape Sequences in Inserted Strings

You can use the following ASCII control characters in strings to be inserted; all other ASCII control characters are ignored.

Escape Sequence Purpose

- \t (or ^I) Tab. (Inserts 1-8 spaces, to bring the text cursor to the next tab stop. Tab stops are every 8 columns.)
- \b (or ^H) Backspace (destructive).
- \n (or ^J) New line (carriage return and line feed).

A new-line character (\n) moves the text cursor to column 1 of the next line, and the inserted text continues at that point. For example, if the textual region contains the following text, and the cursor is at column 4 line 1:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	a	b	c	d	e	f							
2	g	h	i	j	k	l							
3													
4													
5													
6													
7													
8													

and the following text is inserted:

```
add to ABC
insert "Some\ntext"
```

the textual region will display the following:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	a	b	c	S	o	m	e						
2	t	e	x	t	d	e	f						
3	g	h	i	j	k	l							
4													
5													
6													
7													
8													

A backspace character (\b) deletes the character to the immediate left of the text cursor. For example, if the textual region contains the text in the above illustration, when the text cursor is at column 5 line 2, and the following text is inserted:

```
add to ABC
insert "\b\bst"
```


The textual region will display the following:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	a	b	c	S	o	m	e						
2	t	e	s	t	d	e	f						
3	g	h	i	j	k	l							
4													
5													
6													
7													
8													

If you insert a backspace when the text cursor is at column 1, line 1, nothing will be inserted, and the text cursor will not move.

In textual regions, tab stops are every 8 columns; that is, column 9, 17, 25, 33, etc. A tab character (`\t`) inserts 1 to 8 spaces, to bring the text cursor to the next tab stop. For example, if the textual region contains the text in the above illustration (with the text cursor still at column 5 line 2) and the following text is inserted:

```
add to ABC
    insert "\t"
```

the textual region will display the following:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	a	b	c	S	o	m	e						
2	t	e	s	t	^	^	^	^	d	e	f		
3	g	h	i	j	k	l							
4													
5													
6													
7													
8													

8.2.4.3 Inserting a Text Segment

You can copy a segment from one textual region into another, or from one place to another in the same textual region, by using the **segment** specification shown in the **insert** model above.

If both the source and destination textual regions are colored textual regions, **insert segment** will copy the characters' color attributes. If the segment is colored but the destination textual region is not, then the color attributes will be dropped. If the segment is not colored, the default background and foreground colors of the destination textual region will be used.

You can insert a segment from one place to another in the same textual region, as long as the segment does not overlap the current text cursor's position. (If an overlap does occur, no segment will be selected, and an error message will be generated.)

The specified segment is inserted by default beginning at the current position of the text cursor, and the original text is moved to the right to make room for the new text. Note that, as with the **insert** STRING statement, after the text is inserted, the text cursor is positioned on the character following the last inserted character. This is the same character on which the text cursor was positioned before the insertion, but it is now in a new column position. You can optionally specify **after cursor** to indicate that the segment should be placed after the current cursor position.

Line breaks within the segment are preserved when the segment is inserted.

8.2.4.4 Inserting a Text Block

You can copy a text block from one textual region into another, or from one place to another in the same textual region, as long as the block does not overlap the current text cursor's position. (If an overlap occurs, no block will be selected, and an error message will be placed in the errorlog.)

To copy a specified text block, use the **block** specification shown in the **insert** model. The specified block is inserted beginning at the current position of the text cursor, and any text is moved to the right on a line-by-line basis. The lines are not moved down. For example, if the source textual region OldTxt contains the following:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	o	n	e										
2	t	w	o										
3	t	h	r	e	e								
4	f	o	u	r									
5	f	i	v	e									
6	s	i	x										
7													
8													

and the destination textual region NewTxt contains the following, with the text cursor at column 1 line 1 as shown:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	a	b	c										
2	d	e	f										
3	g	h	i										
4													
5													
6													
7													
8													

If the following **insert** statement is executed:

response to ...

add to NewTxt

insert block column 1 line 1 thru column 6 line 6
from OldTxt

NewTxt will look like this:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	a	n	e	^	^	a	b	c					
2	t	w	o	^	^	d	e	f					
3	t	h	r	e	e	^	g	h	i				
4	f	o	u	r									
5	f	i	v	e									
6	s	i	x										
7													
8													

Note that the text cursor does not move. After execution of the **insert** statement, it is at the same column and line position as before—only now it is on the first character of the inserted block.

Also note that some character positions are padded with blanks to form a rectangle. Since a text block is based on a rectangular portion of the textual region, it can include character positions which do not contain any text. These positions are always padded with blanks when the block is inserted. And, unlike a segment, a block can begin anywhere after the last character of the last line in a textual region—EASEL OS/2 EE will simply pad the entire inserted area with blanks.

Like segments, color attributes are preserved when blocks from a colored textual region are inserted into another place in the same region or into another colored textual region.

8.2.4.5 Inserting an ASCII Text File or Color-attributed File

The **insert [colored] file** drawing statement inserts an ASCII text file or color-attributed file into a textual region. The contents of the specified file are inserted starting at the current position of the text cursor.

This statement is exactly the same as the **read** action statement except that, because it is a drawing statement, it allows you to insert a file at runtime *or* compile time. (The **read** action statement inserts a file during runtime only.) After the **insert** statement is executed, the text cursor is on column 1 of the line immediately following the last line of the inserted file.

8.2.4.6 Using the ioerror Function

It is recommended that you test the value of the built-in function **ioerror** when you compile a program that uses the **insert [colored] file** operation, to see if the operation was successful. The value returned by **ioerror** is a boolean value: if the operation was successful (there was no failure), the value is **false**; if it failed, the value is **true**. For example:

```
response to start
  add to Report
    insert file Report
  if ioerror
    then ...
  end if
```

8.2.5 Overwriting Text

The **overwrite** statement, specified according to the model below, lets you replace text in a textual region. You can overwrite text with a text string, a text segment, or a text block.

Model

```
overwrite { [at cursor] [text] STRING |
  segment [column] COL_NO [line] LINE_NO
    [thru [column] COL_NO [line] LINE_NO] from TR_NAME |
  block [column] COL_NO [line] LINE_NO
    [thru [column] COL_NO [line] LINE_NO] from TR_NAME }
```

The **overwrite** statement replaces text with the specified string, text segment, or text block. The replacement begins at the current position of the text cursor, and each character is replaced in succession.

If an **overwrite** statement is executed at a location where there is no text, EASEL OS/2 EE will insert the necessary blanks prior to the text cursor, as it does with the **insert** statement.

8.2.5.1 Overwriting with Text Strings

You can specify a string to overwrite text. For example, if the text cursor is positioned as indicated in the textual region shown below:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1													
2	T	h	i	s									
3													
4													
5													
6													
7													
8													

and you specify:

overwrite "That"

EASEL OS/2 EE overwrites "This" with "That" and creates the text shown below:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1													
2	T	h	a	t									
3													
4													
5													
6													
7													
8													

After execution, the text cursor is on the first character immediately following the last overwritten character.

If the text cursor is at the column after the last character in a line, an **overwrite** STRING statement is performed as if it were an **insert** statement, and the line is extended.

8.2.5.2 Overwriting with Escape Sequences

You can use the following ASCII control characters in strings to be inserted; all other ASCII control characters are ignored.

Escape Sequence Purpose

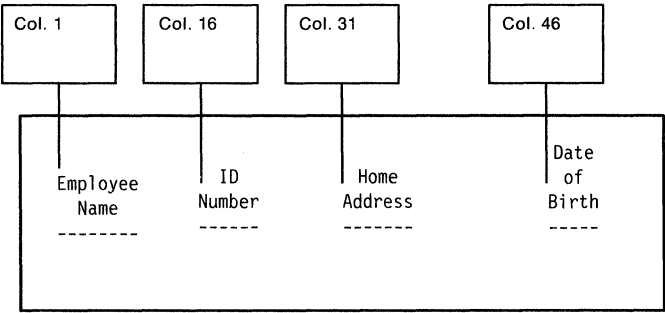
<code>\t</code> (or <code>^I</code>)	(Tab) Inserts 1-8 spaces, to bring the text cursor to the next tab stop. (Tab stops are every 8 columns.)
<code>\b</code> (or <code>^H</code>)	Backspace (destructive).
<code>\n</code> (or <code>^J</code>)	New line (carriage return and line feed).

If you overwrite with a backspace (`\b`) when the text cursor is at column 1, line 1, nothing will be overwritten, and the text cursor will not move.

New-line characters (`\n`) specified in an **overwrite** STRING statement move the text cursor to the next line *at the same column it was in at the time the statement was executed*, and overwriting continues at that point. For example, the following **add to** statement:

```
response to ...
  add to Employee_Data
    move to 1 2
    overwrite "Employee\n Name\n-----"
    move to 16 2
    overwrite " ID\nNumber\n-----"
    move to 31 2
    overwrite " Home\nAddress\n-----"
    move to 46 1
    overwrite " Date\n of\nBirth\n-----"
```

The textual region Employee_Data will display the following:



8.2.5.3 Overwriting with Text Segments

You can overwrite text with a specified text segment, using the **overwrite segment** syntax, shown in the model above. For example:

```
response to ...  
  add to NewTxt  
    overwrite segment 3 1 thru 10 4 from OldTxt
```

After execution, the text cursor is on the character immediately following the last overwritten character.

If the text cursor is at the column after the last character in a line, an **overwrite segment** statement is performed as if it were an **insert** statement, and the line is extended.

Color attributes are preserved in segments from a colored textual region that are used for overwriting text. If the segment is colored but the destination textual region is not, then the color attributes will be dropped. If the segment is not colored but the destination textual region is colored, the characters in the segment inherit the attributes of the text they overwrite.

Line breaks within the segment are retained. That is, when the segment contains more than one line of text, each line overwrites the textual region as if the end of each line was a new-line character (**\n**).

8.2.5.4 Overwriting with Text Blocks

You can overwrite text with a specified text block, using the **overwrite block** syntax, shown in the model above. For example, suppose the source textual region, OldTxt, contains:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	o	n	e										
2	t	w	o										
3	t	h	r	e	e								
4													
5													
6													
7													
8													

and the destination textual region, NewTxt, contains the following text, with the text cursor at column 1 line 1 as shown:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	a	b	c	d	e	f	g	h	i	j			
2	k	l	m	n	o	p	q	r	s				
3	t	u	v	w	x	y	z						
4													
5													
6													
7													
8													

When the following response statement is executed:

```
response to ...  
  add to NewTxt  
    overwrite block column 1 line 1  
      thru column 7 line 4 from OldTxt
```

NewTxt will display the following:³

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	o	n	e	^	^	^	^	h	i	j			
2	t	w	o	^	^	^	^	r	s				
3	t	h	r	e	e								
4													
5													
6													
7													
8													

Note that this statement does *not* move the text cursor.

Like segments, when a block from a colored textual region overwrites other colored text, the block's color attributes are preserved.

³ Remember, the carets shown in the illustrations are not actually displayed.

8.2.5.5 Overwriting Text Blocks with Blanks

You can replace a text block with spaces by specifying a **blank block** drawing statement. Unlike any of the other **overwrite** statements, this statement simply replaces the specified text block with blanks—it does not have a “source” textual region. It also does not affect the position of the text cursor. For example, if a textual region contains the following:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	a	b	c	d									
2	e	f	g	h									
3	i	j	k	l									
4	m	n	o	p									
5	q	r	s	t									
6	u	v	w	x									
7	y	z	a	b									
8	c	d	e	f									

and the following statement is executed:

```
response to ...  
  add to Blackboard  
    blank block column 1 line 1 thru column 1 line 8
```

Blackboard will be changed to the following:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	^	b	c	d									
2	^	f	g	h									
3	^	j	k	l									
4	^	n	o	p									
5	^	r	s	t									
6	^	v	w	x									
7	^	z	a	b									
8	^	d	e	f									

8.3 Response Definitions for Textual Regions

The user selects an object to invoke the **response to** OBJECT definition; thus, a response definition for a textual region, or for a class of textual regions, follows the model shown below.

Model

```
[interrupt] response to { OBJECT_NAME | CLASS_NAME }  
  
    [ACTION_STATEMENT]...
```

For example:

```
response to NewText  
    make NewText background green
```

The senseport (selectable area) of a textual region is the space within, and including, its border.

You can use the **xcoord** and **ycoord** response inquiry built-in functions to ask EASEL OS/2 EE for the exact position of a selection; these functions return the column and line number of the point selected. For example, if a textual region is being used to display a menu of items, with each item on a new line, **ycoord** can allow you to determine which item was selected.

8.4 Action Statements for Textual Regions

8.4.1 Overview

Some of the action statements that can be used with textual regions are applicable to all objects. The list below describes each of these action statements and how it affects a textual region.

Statement	How Statement Affects Textual Regions
add	Adds a new textual region.
add to	Adds new drawing statement(s) without clearing existing ones.
add to class	Adds a textual region to class.
change	Adds new drawing statements and clears existing ones.
change graphics	Adds new drawing statements, and clears existing ones. Although this statement is intended for use with graphical objects that have children, it can be used for a textual region as well. When used, it works exactly like the change statement (since a textual region cannot have children).
change position	Changes position of the textual region's viewport.
clear	Deletes contents of textual region, and moves text cursor to column 1, line 1.
clear graphics	Deletes contents of textual region. This statement, like the change graphics statement, is intended for use with graphical objects that have children, but can be used for a textual region as well. When used, it works exactly like the clear statement (since a textual region cannot have children).
delete	Deletes textual region.
delete from class	Removes textual region from class.

In addition to the statements listed above, there are many action statements unique to textual regions; these are described below. These statements allow you to write a textual region to a file, read a file into a textual region, emphasize text, control the visibility of the text cursor, and otherwise manipulate text. If you specify one of these statements for a graphical object, the action statement will not be executed and an error message will be produced.

8.4.2 Adding Drawing Statements to a Textual Region

The **add to** action statement, adds drawing statements to the contents of a textual region (or to the contents of all the textual regions in a class) in a similar way in which it adds drawing statements to graphical objects. The only difference is that you specify the position in columns and lines, instead of X Y coordinates.

The added contents are specified by the drawing statements in the action statement. For example, the following statement adds the text "memorandum" at column 10, line 4 of the textual region Report, and all the existing drawing statements for the textual region are preserved.

```
add to Report at position column 10 line 4
    insert "memorandum"
```

The **at [position]** specification moves the text cursor to the column and line specified. It is equivalent to specifying a **move to** drawing statement. If you do not include the **at [position]** specification, the new drawing statements begin at the current position of the text cursor.

Because a textual region's drawing statements are distinct from graphical drawing statements, you cannot specify an **add to** statement for a class of objects that includes textual regions as well as other objects.

8.4.3 Removing Text

You can remove one or more lines, a text segment, or a text block from a textual region with the **remove** action statement, shown in the following model.

Model

```
remove { [line] LINE_NO [thru [line] LINE_NO] |
        segment [column] COL_NO [line] LINE_NO
              [thru [column] COL_NO [line] LINE_NO] |
        block [column] COL_NO [line] LINE_NO
              [thru [column] COL_NO [line] LINE_NO] }
from TR_NAME
```

When text is removed, any text following the removed text is moved up and to the left, filling the gap created by the removed text. These statements do not move the text cursor.

You can specify the beginning and ending line, or the beginning and ending of a block or segment in any order. For example, the following are all valid statements:

```
remove line 1 from Writer
```

```
remove line 1 thru line 10 from Writer
```

```
remove line 10 thru line 1 from Writer
```

```
remove segment column 1 line 1 thru column 10 line 1 from  
  Writer
```

```
remove segment column 10 line 1 thru column 1 line 1 from  
  Writer
```

```
remove block column 3 line 1 thru column 8 line 3 from  
  Writer
```

```
remove block column 8 line 3 thru column 3 line 1 from  
  Writer
```

```
remove block column 8 line 1 thru column 3 line 3 from A
```

```
remove block column 3 line 3 thru column 8 line 1 from A
```

```
remove block 3 1 thru 8 3 from Writer
```

8.4.4 Controlling Text Emphasis

The **emphasize** statement emphasizes text in a textual region by reversing the foreground and background colors. For example, if a textual region has a red foreground and a black background, its emphasized text is displayed in black against red. Note that this statement differs from the **highlight** action statement which highlights the entire textual region when it is selected.

The **emphasize** action statement always uses a **segment** specification. For example:

```
response to NewText  
  emphasize segment column 3 line 1  
    thru column 3 line 5 in Reader
```

causes the string of text from column 3, line 1 through column 3, line 5 in NewText to be emphasized as soon as NewText is selected.

You must specify this statement for each textual region you would like to emphasize; you cannot emphasize segments for a class of textual regions. Also, each textual region can have only one segment of emphasized text at

any one time. When a second **emphasize** statement is executed for a particular textual region, the first segment will become de-emphasized.

To explicitly remove the emphasis from a segment, use the **deemphasize** action statement. You can use this statement for a single textual region or for a class of textual regions. In this statement, you need not specify the segment; simply provide the name of the textual region. For example:

```
response to Undo
    deemphasize NewText
```

Text blocks cannot be emphasized with this statement; you will need to use a colored textual region instead.

8.4.5 Changing the Visibility of the Text Cursor

The **make cursor visible** and **make cursor invisible** statements allow you to control the visibility of the text cursor. The text cursor of a textual region is, by default, invisible. When it is made visible, it appears as a rectangle surrounding the character position on which it lies. You can use these statements for a textual region or for a class of textual regions. For example:

```
response to Edit
    make News cursor visible
response to EnterChanges
    make News cursor invisible
```

The text cursor is drawn in the foreground color of the textual region unless the cursor is within an area of emphasized text. For example, if the textual region has a red foreground on a black background, the rectangle appears in red. If the text cursor is within an area of emphasized text, the text appears in reverse video, and the rectangle is therefore displayed in the background color—in this example, black.

8.4.6 Writing a Textual Region to a File

You can write a textual region to a file with the **write** action statement, shown in the following model. You can write a textual region or a colored textual region to an ASCII text file or to a color-attributed file. You can create a new file, replace the contents of an existing file, or append to the contents of an existing file.

Model

```
write [ append | destructive ] TR_NAME to [colored] file FILE_NAME
```

When a textual region is written to a file, only the text is written to the file. If you then read the file back into a textual region, the text assumes the font of

the textual region into which it is copied. None of the **write** statements change the textual region in any way; for example, if it is visible when the **write** statement is executed, it will remain visible.

8.4.6.1 Using the **ioerror** Built-in Function

It is recommended that you test the value of the built-in function **ioerror** after every **write** operation to see if the operation was successful. The value returned by **ioerror** is a boolean value: if the operation was successful (there was no failure), the value is **false**; if it failed, the value is **true**. For example:

```
response to Finished
  write append Chapter1 to file Report
  if ioerror
    then change to program Reformat
  end if
```

The textual region Chapter1 is appended to the text file Report when Finished is selected. Then **ioerror** is tested. If the write operation was not successful, EASEL OS/2 EE changes to the program Reformat.

8.4.6.2 Creating a New File

When you write a textual region to a file without using the keywords **destructive** or **append**, EASEL OS/2 EE creates a new file. For example:

```
write Manual to file "Archive.dat"
```

If a file with the name "Archive.dat" already exists, or if for any other reason the operating system does not let EASEL OS/2 EE create a file by that name, then EASEL OS/2 EE will not perform the operation. An error message will be generated, and the value of **ioerror** will be set to **true**.

8.4.6.3 Appending to a File

You can append the contents of a textual region to the contents of a text file by specifying the keyword **append**. For example:

```
write append Chapter_3 to file Book
```

Here, the contents of the textual region Chapter_3 are appended to the end of the previous contents of the text file represented by the string value Book. If for any reason the operating system will not let EASEL OS/2 EE append the contents of the textual region to the file, EASEL OS/2 EE will not perform the operation, an error message will be generated, and the value of **ioerror** will be set to **true**. If the specified file does not already exist, a new file is created and, in effect, a regular **write** operation is performed.

8.4.6.4 Replacing the Contents of a File

You can replace the existing contents of a text file by specifying the keyword **destructive**. For example:

```
response to ...  
    write destructive FirstDraft to file Store.txt
```

Here, the contents of the textual region `FirstDraft` replace the existing contents of the file `Store.txt`. If you specify a file that does not exist, EASEL OS/2 EE creates a new file, just as if you had not specified **destructive** or **append**. If for any reason the operating system will not let EASEL OS/2 EE delete the old contents of the file, then EASEL OS/2 EE will not perform the operation, an error message will be generated, and the value of **ioerror** will be set to **true**.

8.4.6.5 Writing to a Color-attributed File

You can write a textual region to a color-attributed file by specifying the keyword **colored**. For example:

```
response to ...  
    write Chapter_3 to colored file Book
```

The color-attributed file retains the colors associated with the text and its background. If you write an uncolored textual region to a color-attributed file, the file will retain just the background and foreground colors of the textual region. If you write a colored textual region to a color-attributed file, the file will also retain the colors of individual characters (as defined in a **make segment** or **make block** action statement, or as defined in a color header of a file that was previously read into a colored textual region via the **CText** module).

A color-attributed file is a binary file that assigns foreground and background colors to the text on a character-by-character basis, similar to the way that a word processing program assigns attributes to its text files. *These files cannot be edited with an OS/2 editor.*

When you write a colored textual region to an ASCII text file, the color specifications are not written to the file, and the file can be edited as usual.

8.4.7 Reading a File into a Textual Region

You can read the contents of an ASCII text file or a color-attributed file into a colored or uncolored textual region by specifying a **read** action statement, shown below. This statement works like the **insert file** drawing statement, except that **read** can only be executed at runtime.

```
read [colored] file FILE_NAME into TR_NAME
```

When a **read** statement is executed, the text assumes the font of the textual region into which it is copied, and the text is inserted at the current position of the text cursor. After execution, the text cursor is positioned at column 1 on the line directly following the last line of text from the file. If there is any text following the text cursor prior to the **read** statement, it is moved down to make room for the text from the file.

If the text cursor is at a location where there is no text when the **read** statement is executed, EASEL OS/2 EE will insert the necessary new lines and/or blanks prior to the text cursor, as it does with the **insert** drawing statement.

8.4.7.1 Using the ioerror Function

It is recommended that you test the value of the built-in function **ioerror** after every **read** operation to see if the operation was successful. The value returned by **ioerror** is a boolean value: if the operation was successful (there was no failure), the value is **false**; if it failed, the value is **true**. For example:

```
response to Finished
  read file "Report.txt" into Report
  if ioerror
    then change to program Reformat
  end if
```

When Finished is selected, the file "Report.txt:" is added to the textual region Report, and **ioerror** is tested. If the **read** operation was not successful, EASEL OS/2 EE changes to the program Reformat.

8.4.7.2 Reading an ASCII Text File into a Textual Region

The following statement reads the ASCII text file "Q188sale.txt". into the textual region Money:

```
response to ...
  read file "Q188sale.txt" into Money
```

When you read an ASCII text file into a colored textual region, no special colors are assigned to individual characters. The foreground and background colors will be those specified in the definition of the colored textual region.

8.4.7.3 Reading a Color-attributed File into a Textual Region

The following statement reads the color-attributed file "Info.col" into the colored textual region ColorMoney:

```
response to ...
    read colored file "Info.col" into ColorMoney
```

After execution, the individual characters in the colored textual region are colored as specified in the color-attributed file.

When you read a color-attributed file into an uncolored textual region, the individual character colors are ignored, and the foreground and background colors will be those specified in the textual region's definition. **CText** module

If you are using the **CText** module to assign colors to individual characters in an ASCII text file, do **not** use the **read** action statement; the module uses a special action subroutine to read the file into the textual region.

8.4.7.4 Using Escape Sequences in Files

You can use the following ASCII control characters in files that will be read into a textual region; all other ASCII control characters are ignored.

Escape Sequence	Purpose
<code>\t</code> (or <code>^I</code>)	(Tab) Inserts 1-8 spaces, to bring the text cursor to the next tab stop. (Tab stops are every 8 columns.)
<code>\b</code> (or <code>^H</code>)	Backspace (destructive).
<code>\n</code> (or <code>^J</code>)	New line (carriage return and line feed).

8.4.8 Finding a String of Text in a Textual Region

You can locate text in a textual region, just as most word processors locate specific strings in a file, with the **find** action statement.

Model

```
find in TR_NAME [from [column] COL_NO [line] LINE_NO] STRING
```

If a starting point is not defined with the **from column line** statement, the **find** statement begins searching at the current position of the text cursor, and moves the text cursor to the first character in the first occurrence of the specified string. In the example:

```
response to ...
    find in NewText "Page Two"
```

if the text cursor starts searching at a position after the string "Page Two", the string will not be found.

This statement must be used in conjunction with the **found** built-in function. If the string is not found, the **found** built-in function is set to **false** and the text cursor does not move. For example, if the textual region SearchRegion contains the following:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	C	a	n		y	o	u						
2	f	i	n	d		m	e	?					
3													
4													
5													
6													
7													
8													

When the following response is executed, and the text cursor is at column 1 line 1 of SearchRegion:

```
response to FindIt
    find in SearchRegion "me"
```

The text cursor moves to the "m" in "me", and the following built-in functions are set as shown:

Function	Value
found	true
xcursor	6
ycursor	2

But if the following response is executed:

```
response to FindIt
    find in SearchRegion "her"
```

The value of **found** will be set to **false**, and the text cursor will not move. If **found** returns **true**, you can use the **xcursor** and **ycursor** built-in functions to find out the exact column and line position of the first character of the located string. For example:

```
# Find all occurrences of SearchString in colored textual
# region Document. Highlight each occurrence, using black
# on a red background.
```

```
add to Document
```

```

    move to column 1 line 1
find in Document SearchString
while (found) loop
    copy xcursor of Document to ColumnNo
    copy ycursor of Document to LineNo
    make Document segment ColumnNo LineNo
        thru (ColumnNo + length of SearchString -1) LineNo
        foreground black
        background red
    add to Document
        move by 1 columns 0 lines
    find in Document SearchString
end loop

```

If the cursor is already on the first character of the searched string when the **find** action statement is performed, the built-in function **found** will return **true**, but the text cursor will not move. Thus, simply checking for a change in the values of **xcursor** and **ycursor** after the **find** has been executed is not a reliable way to determine whether or not the string was found.

If you want to search for multiple occurrences of a string, you should move the cursor to the right by one character after each successful **find**; otherwise, an infinite loop will result. This is shown in the above example with the lines:

```

add to Document
    move by 1 columns 0 lines

```

You can specify a location (other than the current cursor position) to begin the search, by specifying **from** and the column and line numbers of the location. For example:

```

response to ...
    find in NewText from column 1 line 30 "Table"

```

finds the first occurrence of the string "Table" starting from column 1, line 30, and moves the text cursor to the "T" in "Table". If "Table" is not found, the cursor does not move.

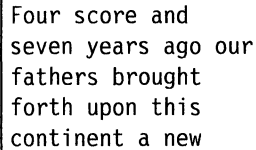
8.4.9 Changing the Size and Position of a Textual Region's Window and Viewport

When using a graphical region, if you change the size of the viewport, the graphical contents of the region's window expand or shrink to completely fill the viewport. In a textual region, the contents cannot expand or shrink, because the displayed size of the text is a function of the font in effect, and the size of the text characters of a font cannot be changed. Therefore, when you change the size of the viewport, characters do not get bigger or smaller; rather, more or less text is displayed to fill the viewport.

Suppose you have a textual region defined as follows:

```
textual region News window size 19 columns 5 lines  
  at position 100 100  
  font "large"
```

The window size of the textual region is specified in terms of the number of columns and lines of text that will fit in the window; the size of the viewport is inferred to match the size of the window. This textual region is shown below:



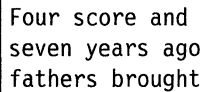
Four score and
seven years ago our
fathers brought
forth upon this
continent a new

Suppose you reduce the size of the window; for example:

response to ...

```
change News window size to 15 columns 3 lines
```

The size of the viewport is reduced automatically. The new viewport is shown below:



Four score and
seven years ago
fathers brought

8.4.9.1 Changing the Viewport Size

You can change the size of the viewport of a textual region (or of all the textual region viewports in a class) with the **change size** action statement in the same way in which a graphical region's size is changed (in X Y coordinates of its parent).

Note: It is recommended that you do not define or change a textual region's viewport size with the **change size** statement, unless you want it to occupy a given size relative to its parent.

It is important to note the visual effects produced by changing the size of a textual region's viewport. When you change the viewport size, the window size also changes (and vice versa), because text does not scale. Thus, a different number of lines and columns will be displayed in the changed textual region.

If you change the window or viewport size of the textual region's *parent*, the textual region is affected differently, depending on whether you defined the textual region by viewport size (**size**) or by **window size**. If you defined the textual region by the **size** specification, the textual region and its window will be changed accordingly. But if you defined the textual region by the **window size** specification, the appearance of the textual region will not change; it will still contain the same number of columns and lines.

8.4.9.2 Changing the Viewport Position

The position of a textual region's viewport is changed with the **change position** action statement, described in 6.3.5.2, "Changing Position" on page 6-31.

8.4.9.3 Changing the Window Size

You can change the size of the window of a textual region (or of all the textual region windows in a class) with the **change window size** action statement. This is the same statement used to change the window size of a graphical region; the only difference is that you must specify columns and lines rather than X Y coordinates. This is because a textual region's window is specified in the textual region's coordinate system, and not that of its ancestor.

If you change the viewport or window size of the textual region's *parent*, the window size of the textual region will change *only* if you defined or changed the textual region's size by viewport size (in X Y coordinates of the textual region's parent).

The **change window size by** statement changes the relative size of the window by positive or negative integer values representing the number of columns and lines by which the size is to change. For example:

response to ...

```
change Comment window size by 30 columns 10 lines
```

If Comment's window size had been 40 columns wide and 60 lines long, it will be 30 columns wider and 10 lines longer after this statement is executed (a total of 70 columns by 70 lines). The top left of the textual region has moved, but still contains the same character (column 1, line 1). The bottom left of the textual region has not moved—it is at the same X Y coordinates of its parent—but now contains new text (or blanks).

If you use negative values, the result must be non-negative. For example:

response to ...

```
change Comment window size by 30 columns -60 lines
```

If Comment's window size had been 40 columns by 50 columns, the result would be a total of 70 columns by -10 lines, which is invalid.

The **change window size to** statement changes the absolute size of the window with positive integer values representing the number of columns and lines. These represent the new dimensions of the window. For example:

```
response to ...  
  change Box window size to 30 columns 50 lines
```

No matter what size Box's window was, it will be 30 columns wide and 50 lines long after this statement is executed.

8.4.9.4 Changing the Window Position

You can change the position of the window of a textual region (or of all the textual region windows in a class) with the **change window position** action statement. This is the same statement used to change the window position of a graphical region; the only difference is that you must specify columns and lines rather than X Y coordinates.

The **change window position by** statement changes the relative position of the window by the values specified. For a textual region, the values are specified in positive or negative integer values representing the number of columns and lines. For example:

```
response to ...  
  change Clipboard window position by 30 columns 5 lines
```

If Clipboard's window position had been at column 40, line 20, it will be at column 70, line 25 after this statement is executed.

The **change window position to** statement changes the absolute position of the window with positive integer values representing the number of columns and lines. These represent the new position of the window within the textual region's coordinate system, which is the upper-left corner of the window. For example:

```
response to ...  
  change Box window position to column 30 line 50
```

No matter where Box's window was positioned, it will be at column 30, line 50 after this statement is executed. When the window has moved, the specified column and line number are at the upper left corner of the window.

The **change window position by** and **change window position to** statements are often used for scrolling through a textual region. Suppose only a portion of the contents of the textual region is visible in its window. You will want the user to be able to select an up-arrow key to view the contents that are above the top of the window, and to select a down-arrow key to view the contents that are below the bottom of the window. For example:

```
response to UpArrow
    change DataArea window position by 0 -10
```

```
response to DownArrow
    change DataArea window position by 0 10
```

The **change window position to** statement moves the window so that the specified column and line number are at the upper left corner of the window. The **change window position by** statement moves the window the specified number of columns and lines. A positive value for columns and lines scrolls the window right and down, respectively; a negative value for columns and lines scrolls the window left and up, respectively.

8.5 Using Built-in Functions with Textual Regions

8.5.1 Overview

The built-in functions that can be used with textual regions as well as other objects are shown below. They return different values for textual regions than they do for other objects, since the values are based on lines and columns rather than X Y coordinates.

Function	Value Returned When Applied to Textual Regions
top [of]	Line number of the first line (always 1) of the textual region.
bottom [of]	Line number of the last line of text in the textual region.
left [of]	Column number of the leftmost column (always 1) of the textual region.
right [of]	Column number of the rightmost character in the longest line of the textual region.
xmiddle [of]	The value of right [of], divided by 2.
ymiddle [of]	The value of bottom [of], divided by 2.
xcoord	Column number of the character selected.
ycoord	Line number of the character selected.
xcursor [of]	Column number of the position of the text cursor.
ycursor [of]	Line number of the position of the text cursor.
xposition [of]	X coordinate of the lower left corner of the viewport, in the parent's coordinates.
yposition [of]	Y coordinate of the lower left corner of the viewport, in the parent's coordinates.
xsize [of]	The X dimension of the textual region's size, in its parent's coordinate system. For textual regions defined by window size , this is the number of columns plus 1, multiplied by the column size, in the textual region's coordinates.

Function	Value Returned When Applied to Textual Regions
ysize [of]	The Y dimension of the textual region's size, in its parent's coordinate system. For textual regions defined by window size , this is the number of lines plus 1, multiplied by the line size, in the textual region's coordinates.
ancestry [of]	String associated with the textual region's ancestry.
background at	Integer that is the background color of the specified line and column of text.
background of	Integer associated with the background color of the textual region. (Note that this is different from the background at function.)
border of	Integer associated with the color of the textual region's border.
exists	Boolean associated with the textual region's name.
foreground at	Integer that is the foreground color of the specified line and column of text.
foreground of	Integer associated with the foreground color of the textual region. (Note that this is different from the foreground at function.)
handle of	Integer that is the PM handle for the object.
line size of	Height of line, in the parent's coordinates.
parameter of	Parameter string associated with the specified textual region.
parameter	Parameter string associated with the selected textual region.
priority of	Integer associated with the priority of the textual region.
selectability of	Boolean associated with the selectability of the textual region.
text of	String containing the title bar text.
textual { line LINE_NO SEGMENT_DEF BLOCK_DEF } from TR_NAME	String containing the contents of the specified line, segment, or block.
visibility of	Boolean associated with the visibility of the textual region.
window xsize [of]	Width of the window, in lines.

Function	Value Returned When Applied to Textual Regions
window ysize [of]	Height of the window, in columns.
window xposition [of]	Column number of the leftmost character in the window.
window yposition [of]	Line number of the topmost character in the window.
found	True if the result of a find statement was successful; false otherwise.
ioerror	True if the result of a read or write action statement or an insert file drawing statement was unsuccessful; false otherwise.
object	String containing the name of an object or item for which a response is taken.

In addition to the built-in functions listed above, there are many built-in functions unique to textual regions. Some of these have already been introduced in this chapter; each one is described below in detail.

8.5.2 Using the textual Built-in Functions

Each of the the **textual** built-in functions returns a string value containing the contents of a specified line, segment, or block from a textual region. The most common use of these functions is to copy the specified contents to a string value.

Model

```

textual { line LINE_NO |
           segment [column] COL_NO [line] LINE_NO
               [thru [column] COL_NO [line] LINE_NO] |
           block [column] COL_NO [line] LINE_NO
               [thru [column] COL_NO [line] LINE_NO] } from TR_NAME

```

8.5.2.1 The textual line Function

When you inquire about a line, the value returned is the string of text found on the specified line number of the specified textual region. For example, if you define a textual region and put text into the region as follows:

```

textual region Manual size 400 400 at position 10 10
    insert "This is line one\n This is line two"

```

and later specify in the program:

```
response to ...  
  copy textual line 1 from Manual to CurrentLine
```

EASEL OS/2 EE copies the following string value to CurrentLine:

"This is line one"

Note that the string does *not* include the new-line sequence (`\n`).

8.5.2.2 The textual segment Function

When you inquire about a segment, the value returned is the string containing the contents of the specified segment. For example, if the source textual region Speech containing the following:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	F	o	u	r		s	c	o	r	e			
2	a	n	d		s	e	v	e	n				
3	y	e	a	r	s		a	g	o		o	u	r
4	f	o	r	e	f	a	t	h	e	r	s		
5	b	r	o	u	g	h	t	.	.	.			
6													
7													
8													

And the following response statement is invoked:

```
response to Speech  
  copy textual segment column 6 line 1  
    thru column 9 line 3 from Speech to Excerpt
```

The following string would be copied to the string variable Excerpt:

"score\nand seven\nyears ago"

When the **insert STRING** drawing statement contains the built-in function **textual segment** and it is used with a colored textual region, it drops any character-oriented color information from the inserted segment. For example, in the following statement, any color attributes in the segment are retained:

```
response to ...  
  add to ColoredNewText  
    # The following inserts a segment  
    # with color attributes:  
    insert segment column 25 line 6  
      thru column 16 line 7 from ColoredOldText
```

But in the following statement, any color attributes are dropped, and the segment is drawn in the foreground and background colors of the textual region ColoredNewText:

```
response to ...
  add to ColoredNewText
    # The following inserts a string returned by the
    # textual segment function. Color attributes are
    # dropped:
    insert textual segment column 25 line 6
      thru column 16 line 7 from ColoredOldText
```

8.5.2.3 The textual block Function

The **textual block** function returns a string containing each line within the specified block, separated by a single blank character. This inquiry makes it easy to copy a column of numbers from a textual region into a single string. For example, if the text block in the textual region contains the following:

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	^	1	2	3	.	4	5						
2	8	8	7	7	.	0	3						
3	^	-	9	8	.	7	6						
4	^	4	2	2	.	6	2						
5													
6													
7													
8													

and the program contains the following statement:

```
copy textual block column 1 line 1
  thru column 8 line 4 from OldText to BlockVar
```

the following string would be copied to the variable BlockVar:

```
" 123.45 8877.03 -98.76 422.62 "
```

Be careful when using the **textual block** function with the **insert STRING** drawing statement. What is inserted is not a block, but a string returned by the **textual block** function. The string contains the text from each line within the specified block. For example, the following statement inserts a block into NewText:

```
response to ...
  add to NewText
    insert block column 10 line 4 thru column 50 line 8
      from OldText
```

But the following statement, which uses the **textual block** function, inserts the *string* returned by **textual block**:

```
response to ...
  add to NewText
    insert textual block column 10 line 4
      thru column 50 line 8 from OldText
```

8.5.2.4 Extracting from a Textual Region

You can use the **textual** built-in functions with the **extract from** action statement to extract a string value from a textual region and copy it to a variable. The full syntax for this statement uses the following model.

Model

```
extract from
textual { line LINE_NO |
  segment [column] COL_NO [line] LINE_NO
    [thru [column] COL_NO [line] LINE_NO] |
  block [column] COL_NO [line] LINE_NO
    [thru [column] COL_NO [line] LINE_NO] } from TR_NAME

[SKIP_CLAUSE]...
TAKE_CLAUSE
[ TAKE_CLAUSE | SKIP_CLAUSE]...
```

See 4.5.3, “The extract from Action Statement” on page 4-22 for details on the use of the **extract from** statement, including its associated SKIP_CLAUSE and TAKE_CLAUSE.

8.5.3 Using the column size of Built-in Function

The **column size** of built-in function returns the width of a column, in the coordinates of the textual region's parents. The value returned is an integer representing the X width of the column in the specified textual region. The width of the column is the width of a character plus the character's associated horizontal intercell gap.

In the following example, the disabled key Logo is positioned in Product's parent such that it is visually just to the right of Product's text cursor:

```
green textual region Product size 100 100
  at position 0 0 in CoData
```

```
response to ...
  change Logo in CoData position to
    ((xcursor of Products) * column size of Products) 0
```

8.5.4 Using the line size of Built-in Function

The value returned by the **line size of** function is an integer representing the Y height of a line in the specified textual region, in its parent's coordinate system. A line is the height of a character, plus the character's associated vertical intercell gap. The value returned is in the parent's coordinates.

In the example below, the disabled key Logo is positioned in N's parent such that it is visually just below the last line of text in N:

```
green textual region N size 100 100
  at position 10 10 in CoData
  :

response to ...
  change Logo in CoData position to
    0 (yposition of N+(ysize of N-(ycursor of N*line
      size of N)))
```

8.5.5 Using the foreground at and background at Built-in Functions

The **foreground at** and **background at** functions are used to find out the current foreground or background color of a specific character position in a colored textual region. They can be used with non-colored textual regions as well; in this case, they will return the same values returned by the **foreground of** and **background of** functions.

Model

```
{ foreground | background } at [column] COL_NO [line] LINE_NO char
  of TR_NAME
```

Each function returns an integer value associated with the color of the background or foreground of the character at the specified line and column. For example:

```
response to News
  copy foreground at column 1 line 3 char
    of InfoWindow to ForeColor
  copy background at column xcoord line ycoord char
    of News to SelectedColor
```

8.5.6 Using the *found* Built-in Function

The value returned by the **found** built-in function is a boolean associated with the existence of the specified string in the textual region. It is used to test the success of the **find** action statement. If the string was found, the value of **found** is **true**; if the string was not found, the value of **found** is **false**.

8.6 Sample Programs Using Textual Regions

8.6.1 Sample #1

This program displays a textual region. The textual region is initially empty. When the user types on the keyboard, the typed characters are displayed in the textual region at the position of the text cursor.

The user can move the text cursor by selecting anywhere within the textual region's window. The cursor will move to the character position selected.

An EXIT key is also defined and displayed. When the user selects this key, EASEL OS/2 EE terminates.

Following is the definition of the textual region. Its window is 24 # columns wide and 12 lines long and there is a border around it:

```
enabled blue textual region TextReg
  window size 24 columns 12 lines
  at position 100 100
  white border
  font "large"
```

Following is the definition for the EXIT key. It is displayed as a box # with the word "EXIT" centered inside it:

```
red key ExitKey at position 600 100
  box 100 80
  move to 50 40
  font "large" text center "EXIT"
```

The following response definition causes the text cursor to be visible # when EASEL OS/2 EE begins:

```
response to start
  make TextReg cursor visible
```

The following response definition causes the text cursor to move to the # point selected when the textual region is selected:

```
response to TextReg
  add to TextReg
    move to column xcoord line ycoord
```

```
# The following response definition causes a character, typed from the
# keyboard, to be inserted into the textual region at the position of the
# text cursor:
```

```
response to char from keyboard
    add to TextReg
        insert input
```

```
# The following response definition causes EASEL OS/2 EE to terminate
# when the EXIT key is selected:
```

```
response to ExitKey
    exit
```

8.6.2 Sample #2

In this example, two textual regions are presented. The one on the left side of the screen, SourceTextReg, contains text which the user can transfer to the one on the right side of the screen, DestTextReg.

To accomplish a transfer of text, the user first selects the yellow keys to choose between doing an **insert** or an **overwrite** operation and between a **segment** or **block**. Each time a yellow key is selected, it toggles between one of its two states.

Next, the user selects two endpoints in SourceTextReg. When the first endpoint is selected, the cursor becomes visible and is moved to the selected character position. When the second endpoint is selected, the cursor becomes invisible again, but the entire segment or block defined by the two endpoints is highlighted with a red background.

Once the text to be transferred is selected, the user selects a character position within DestTextReg and the text is inserted or overwritten starting at that location. The selected text can be inserted or overwritten multiple times at various character positions within DestTextReg. The user can select other segments or blocks in SourceTextReg at any time.

The green "CLEAR" key, when selected, clears DestTextReg. The red "EXIT" key causes EASEL OS/2 EE to exit.

```
# VARIABLES
```

```
boolean variable
    DoInsert is true
    DoSegment is true
```

```
integer variable
    StartColumn
    StartLine
    StopColumn
```

StopLine

OBJECTS

enabled blue colored textual region SourceTextReg

window size 13 columns 6 lines

at position 100 250

white border

font "large"

insert "Here is some\n"

insert "text for you\n"

insert "to transfer.\n"

insert "1234567890+-\n"

insert "ABCDEFGHJKLMN\n"

insert "NOPQRSTUVWXYZ\n"

disabled blue textual region DestTextReg

window size 18 columns 9 lines

at position 400 250

white border

font "large"

yellow key InsertKey at position 100 100

box 100 80

move to 50 40

font "medium" text center "INSERT"

invisible yellow key OverwriteKey at position 100 100

box 100 80

move to 50 40

font "medium" text center "OVERWRITE"

yellow key SegmentKey at position 250 100

box 100 80

move to 50 40

font "medium" text center "SEGMENT"

invisible yellow key BlockKey at position 250 100

box 100 80

move to 50 40

font "medium" text center "BLOCK"

green key ClearKey at position 450 100

box 100 80

move to 50 40

font "large" text center "CLEAR"

red key ExitKey at position 650 100

box 100 80

```

    move to 50 40
    font "large" text center "EXIT"

# RESPONSES

response to InsertKey
    make InsertKey invisible
    make OverwriteKey visible
    copy false to DoInsert

response to OverwriteKey
    make OverwriteKey invisible
    make InsertKey visible
    copy true to DoInsert

response to SegmentKey
    make SegmentKey invisible
    make BlockKey visible
    copy false to DoSegment

response to BlockKey
    make BlockKey invisible
    make SegmentKey visible
    copy true to DoSegment

response to ClearKey
    clear DestTextReg

response to ExitKey
    exit

response to SourceTextReg
    disable DestTextReg
    make SourceTextReg background blue
    add to SourceTextReg
        move to column xcoord line ycoord
    make SourceTextReg cursor visible
    copy xcoord to StartColumn
    copy ycoord to StartLine
    begin guarded
        response to SourceTextReg
            make SourceTextReg cursor invisible
            copy xcoord to StopColumn
            copy ycoord to StopLine
            if (DoSegment) then
                make SourceTextReg segment
                column StartColumn line StartLine thru
                column StopColumn line StopLine
                background red

```

```

        else
            make SourceTextReg block
                column StartColumn line StartLine thru
                column StopColumn line StopLine
            background red
        end if
        enable DestTextReg
        leave block
    end

response to DestTextReg
    add to DestTextReg
        move to column xcoord line ycoord
    if (DoInsert) then
        if (DoSegment) then
            add to DestTextReg
                insert segment StartColumn StartLine thru
                StopColumn StopLine from SourceTextReg
        else
            add to DestTextReg
                insert block StartColumn StartLine thru
                StopColumn StopLine from SourceTextReg
        end if
    else
        if (DoSegment) then
            add to DestTextReg
                overwrite segment StartColumn StartLine thru
                StopColumn StopLine from SourceTextReg
        else
            add to DestTextReg
                overwrite block StartColumn StartLine thru
                StopColumn StopLine from SourceTextReg
        end if
    end if
end if

```

8.7 Working with Colored Textual Regions

A textual region can also be defined as **colored**. In a colored textual region, you can specify background and foreground colors for individual characters.

All drawing statements and action statements for uncolored textual regions can be used with colored textual regions. Colored textual regions have extra, unique action statements to specify colors for individual characters.

8.7.1 Defining a Colored Textual Region

You can make a textual region colored simply by adding the keyword **colored** to the definition, see 8.1.2, "Defining a Textual Region" on page 8-3.

The color specifications in the definition are for the background and foreground of the *entire* region, exactly like an uncolored textual region. The color specifications for *individual characters* in the region override the color specifications in the definition. (The color attributes are preserved when you use the **save program as** statement.) Character locations which do not contain any text are displayed in the background color specified in the definition of the colored textual region.

8.7.1.1 Ways to Color Text

After defining the textual region as **colored**, you can specify the position and color for any of its characters in one of two ways (or a combination of both):

- If text already exists in the colored textual region in your program, use the **make segment** or **make block** action statement.
- If you have an ASCII text file that will be read into the colored textual region, use the **CText** Module.

When you assign a color to a character in a colored textual region, you assign it to the character on a column and line basis, *but the color is assigned to the character itself, not to the character's position*. Therefore, if the character changes position (for example, if text is inserted or deleted during runtime), it will retain its assigned background and foreground colors. The colors are *not* applied to text which, when inserted or read in, occupies that portion of the textual region. But if you **overwrite** non-colored text into an area that has been colored, the new text *will* take on the color attributes of the overwritten characters.

When colored text is emphasized (see 8.4.4, "Controlling Text Emphasis" on page 8-29), the foreground and background colors of each of the characters within the emphasized segment are reversed.

8.7.2 Coloring Text That Already Exists in the Program

The **make segment** and **make block** action statements allow you to assign colors to individual characters of existing text in a colored textual region. For text that is to be read into the colored textual region from an ASCII text file during runtime, you can use the **CText** module to assign colors to the individual characters. Note that attempts to color individual characters in an uncolored textual region will be ignored, and an error message will be produced.

The color specifications in the **make segment** and **make block** action statements are applied to the colored textual region on a character-by-character basis. You can color just one character, or one group of characters, or several groups of characters. Characters *not* colored with this statement will be displayed in the background and foreground colors specified in the definition of the colored textual region.

The **make segment** and **make block** action statements use the following model. Note that if you change the foreground or background color of the entire colored textual region with the **make COLOR** action statement, *all* text will inherit the new foreground color, even those characters that had been individually colored with the **make segment** or **make block** statements. Likewise, all such text will inherit the newly specified background color.

Model

```
make CTR_NAME { block | segment } [column] COLUMN_NO [line] LINE_NO
               [thru [column] COLUMN_NO [line] LINE_NO]
               [foreground [color] COLOR]
               [background] [color] COLOR
```

8.7.2.1 Specifying Colors for a Text Segment

The **make segment** action statement specifies a segment the same way a segment is specified in other statements. For example, if the statement reads:

```
response to ...
  make Region_1 segment column 1 line 1
    thru column 5 line 3
    foreground red
```

and Region_1 contained the following text, the portion that would be the foreground color red is highlighted below:

	1	2	3	4	5	6	7	8	9
1	T	h	i	s	i	s	a		
2	a	t	e	s	t	o	f		
3	c	o	l	o	r	e	d		
4	t	e	x	t	.				
5									
6									
7									
8									

To color two or more text segments in the same colored textual region, you must write a series of **make segment** statements, such as:

```
response to ...
  make Region_6 segment column 1 line 1
    thru column 40 line 1 red
  make Region_6 segment column 1 line 5
    foreground color 5 color 24
  make Region_6 segment column 1 line 8
    thru column 40 line 10 blue
```

8.7.2.2 Specifying Colors for a Text Block

The **make block** action statement specifies a block the same way a block is specified in other statements. For example, if the statement reads:

```
response to ...
  make Region_1 block column 1 line 1
    thru column 5 line 3
    foreground red
```

and Region_1 contained the following text, the portion that would be the foreground color red is highlighted below:

	1	2	3	4	5	6	7	8	9
1	T	h	i	s	i	s	a		
2	t	e	s	t	o	f			
3	c	o	l	o	r	e	d		
4	t	e	x	t	.				
5									
6									
7									
8									

Note that the padded blanks in column 5 will be the default background color of the colored textual region, unless their background color was modified by any previous **make segment** or **make block** statements).

To color two or more text blocks in the same colored textual region, you must write a series of **make block** statements, such as:

response to ...

```
make Region_6 block column 1 line 1
    thru column 40 line 1 red
make Region_6 block column 1 line 5
    foreground color 5 color 24
make Region_6 block column 1 line 8
    thru column 40 line 10 blue
```

8.7.3 Specifying Colors for Text in a File: Using the CText Module

The **CText** module allows you to specify background and/or foreground colors for individual characters in an ASCII text file as that file is read into a colored textual region during runtime. If the text that you want to color already exists in the colored textual region in your program, use the **make segment** or **make block** action statement instead of the **CText** module.

8.7.3.1 Summary of Procedures

The **CText** module is automatically installed during the EASEL OS/2 EE installation process. The following steps summarize the use of the **CText** module:

1. Edit the ASCII text file that you want to read into a colored textual region, by adding a color header at the top of the file.
2. Edit your EASEL OS/2 EE program as follows:
 - a. Reference the module at the top of your program.
 - b. Create a colored textual region and an uncolored textual region.
 - c. Copy the name of the colored textual region into the **CText** module's string variable **CTR_Region**. This colored textual region will hold the colored text.
 - d. Copy the name of the uncolored textual region into the **CText** module's string variable **CTR_Header**. This textual region will hold the color specifications and comment lines from the color header in the ASCII text file.
 - e. Copy the name of the ASCII text file that will be read into the colored textual region into the **CText** module's string variable **CTR_File**.
 - f. Invoke the **CText** module's action subroutine **CTR_Read**.

8.7.3.2 Specifying Text Colors in an ASCII File

Before using the **CText** module, you must edit the ASCII text file that you want to read into a colored textual region, adding a color information header at the top of the file. The module will read the information in the color header and apply it to each specified character in the file.

When the text is read into a colored textual region using the **CText** module, the color header will be stripped off and will not appear in the colored textual region. The color specifications and the comment lines in the color header will be inserted into the textual region specified by the predefined variable **CTR_Header**.

The model for specifying text colors in an ASCII text file is shown in the following model.

Model

```
###BeginColors
[default FOREGROUND_COLOR BACKGROUND_COLOR]
[comment COMMENT_LINE]
COLOR_SPECIFICATION
[COLOR_SPECIFICATION]...
###EndColors
```

COLOR_SPECIFICATION uses the model:

Model

```
COLUMN_NO LINE_NO COLUMN_NO LINE_NO
  FOREGROUND BACKGROUND [ segment | block ]
```

Note that the keywords **column** and **line** *cannot* be used in these statements. (They are optional in other **block** and **segment** specifications.)

When determining the column and line numbers of the text to be colored, keep in mind that the module strips out the color header from the ASCII text file when it reads it to the colored textual region. This means that column 1 of the line *immediately* following the **###EndColors** line is equivalent to column 1, line 1 of the text. (You may wish to determine the column and line numbers prior to adding the color header, to avoid confusion.)

In cases where color specifications overlap an area of text, the color specifications that appear *later* in the color header take priority over previous color specifications.

8.7.3.3 Specifying Default Colors

You can add the **default** keyword to the color header, which assigns a background and foreground color to all characters that are *not* within any of the specified color areas. This default overrides the defaults specified in the definition of the colored textual region. The **default** specification can be placed anywhere in the color header, but it is recommended that you place it on the line immediately following the **###BeginColors** line. For example:

```
###BeginColors
default blue white
1 1 5 1 red white segment
###EndColors
```

The color header uses only one **default** specification for the entire file. (If more than one is specified, the last one specified is the one that is used.) If no **default** specification is present in the color header, or if the file has no color header at all, the **CText** module will display the text colors in the background and foreground colors specified in the definition of the colored textual region.

8.7.3.4 Adding Comments to the Color Header

You can add comments to the color header by beginning a line with the keyword **comment**. The color header can contain any number of comment lines. For example:

```
###BeginColors
comment Heading #1 (Company name) is green on blue
3 1 3 6 green blue segment
comment Heading #2 (Company address) is red on blue
5 3 5 6 red blue segment
###EndColors
```

The module places these comments lines, along with the color specifications, in the textual region that is specified by the predefined variable **CTR_Header**.

8.7.3.5 Specifying a Block or Segment of Colored Text

When **block** is specified, EASEL OS/2 EE will color just the text that appears within the specified lines and columns. When **segment** is specified, EASEL OS/2 EE will color all contiguous characters inclusive of the beginning and ending positions. For a complete description of text segments and blocks, see 8.2.3, "Defining Text Segments and Text Blocks" on page 8-10.

8.7.3.6 Using the CText Module

After you have added the color header to your ASCII text file, you can use the **CText** module.

The **CText** module reads the information in the color header and applies the specified colors to the individual characters through EASEL OS/2 EE commands in the module. Both the color header and the text in the ASCII text file can be edited at any time and read into the colored textual region without any changes to the program.

If you apply the **CText** module to an ASCII text file that does *not* have a color header, the text will be displayed in the foreground and background colors specified in the definition of the colored textual region.

Follow these steps to use the **CText** module:

1. Reference the module at the top of your program by inserting the following environment declaration:

```
module CText
```

Note that you do not need to install the **CText** module; this was done automatically when you installed EASEL OS/2 EE.

2. Create a colored textual region. This region will hold the colored text from the ASCII text file.
3. Create an uncolored textual region, to hold the comment lines and the color attribute lines from the color header in the ASCII text file.
4. Copy the name of the colored textual region into the predefined string variable **CTR_Region**, using the following model:

```
copy COLORED_TEXT_REGION_NAME to CTR_Region
```

For example:

```
copy News to CTR_Region
```

5. Copy the name of the uncolored textual region into the predefined string variable **CTR_Header**, using the following model:

```
copy TEXT_REGION_NAME to CTR_Header
```

For example:

```
copy NewsHeader to CTR_Header
```

This textual region is only used to hold the comment lines and the color specification lines from the color header in the ASCII text file. It does not hold the **default** specification or the command lines that begin and end the color header. If you are storing information in the **comment** lines in the color header, or if you want to find out the color specifications for the file, you can access them by extracting lines from this textual region.

6. Copy the name of the ASCII text file into the predefined string variable **CTR_File**, using the following model:

copy "ASCII_FILE_NAME" to CTR_File

For example:

copy "news.dat" to CTR_File

Note that if the file does not exist, if the incorrect file name is given, or if the file is read-protected, the built-in function **ioerror** will be set to **true** and an error message will be generated.

7. Specify the predefined action subroutine **CTR_Read** exactly as follows, as part of a response definition:

action CTR_Read

The ASCII text file will be read into the colored textual region, always beginning at column 1, line 1, with text displayed in the colors specified in the file's color header. Any text that already exists in the region will be moved down.

8.7.3.7 Sample Program Using the CText Module

This sample program uses the sample ASCII text file "sales.txt," which is shown in its entirety directly following the program.

```
screen size 640 350                                # FIRST LINE OF PROGRAM

module CText

invisible textual region Header_Info                # Since this region is
  window size 1 columns 1 lines                     # not displayed, it can
  at position 0 0                                    # be minimum size.
                                                    #
colored textual region Sales_Data                   # Note that this textual
  window size 76 columns 10 lines                   # region contains the
  at position 0 50                                   # keyword "colored." The
  white border                                       # color-attributed file
  font "medium"                                     # "sales.txt" will be
                                                    # read into it.

response to start
  copy "Header_Info" to CTR_Header
  copy "Sales_Data" to CTR_Region
  copy "sales.txt" to CTR_File
  action CTR_Read

response to char "exit" from keyboard
  exit                                              # LAST LINE OF PROGRAM
```

8.7.3.8 Sample ASCII Text File “sales.txt”

```
###BeginColors
comment- Gross Sales Volume for 1986
comment- The default colors will appear in white on
comment- a blue background
comment- Negative values will appear in white on a red
comment- background
default white blue
48 6 49 6 white red
52 6 54 6 white red
17 7 19 7 white red
23 7 24 7 white red
43 7 44 7 white red
33 8 34 8 white red
58 8 59 8 white red
37 9 39 9 white red
43 9 44 9 white red
63 6 64 9 white red block
###EndColors
```

GROSS SALES VOLUME FOR FISCAL 1986
(in Millions)

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Northern	21	13	17	15	10	7	-5	-12	2	-3	9	17
Central	-11	-3	4	9	6	-1	10	9	8	-4	7	10
Southern	13	15	7	-6	10	3	2	11	-6	-8	4	2
Western	8	4	7	10	-11	-3	4	9	6	-1	10	9

8.7.3.9 Public File

The following text is from the file named **CTEXT.PUB** in the
\\EASEL-EE\MODULES directory.

public body CText is

```
string variable  CTR_Header  # Textual region where the
                        # color header is copied.
                  CTR_Region  # Textual region where the
                        # text will be placed.
                  CTR_File    # Name of the ASCII file
                        # to be read in.
```

```
# Action to read the file CTR_File into the region
# CTR_Region, apply colors specified in the header,
# and copy the color specifications (and any comment
# lines) to CTR_Header:
```

```
action CTR_Read is
    action %CTR_Read
```

end public body CText

Chapter 9. Dialog Boxes, Dialog Regions, and Dialog Controls

9.1 Dialog Boxes

A dialog box is an EASEL OS/2 EE object used to request information from the user. A dialog box can be moved but its size cannot be changed. There are two types of dialog boxes:

- Modal
- Modeless

When a modal dialog box is made visible, no other EASEL OS/2 EE objects can be selected until the box is made invisible or deleted. A modeless dialog box allows the user to interact with other visible objects.

Dialog boxes have no ancestry and are made up of any combination of dialog control objects (see 9.3, “Dialog Control Objects” on page 9-7).

A dialog box can have only dialog controls as children.

Dialog boxes are defined in dialog units. Dialog units are proportional to the size of the Presentation Manager (PM) system font. A dialog unit is defined to be one-quarter of the average width and one-eighth of the maximum height of the system font.

9.1.1 Defining Dialog Boxes

The initial attributes of a dialog box are defined in the **dialog box** statement. You can change any of these attributes during runtime except for the following:

- border type and visibility
- title bar visibility
- system menu visibility and system menu contents

The definition uses the following model.

Model

```
[ enabled | disabled ] [ visible | invisible ]  
[ modal | modeless ] dialog box DB_NAME  
  size WIDTH HEIGHT  
  at [position] X Y  
  [parameter is PAR_STRING]  
  [ [dialog] border]  
  [title bar TITLE_STRING]  
  [system menu]
```

When specifying attributes, the **parameter**, **border**, **title bar**, and **system menu** can be specified in any order. Other attributes must be specified in the order shown in the model.

Only the **dialog box** keywords, the identifier, the **at** [**position**] specification, and the **size** specifications are required. Other attributes are optional, using the defaults shown below.

Default Values

SELECTABILITY:	enabled
VISIBILITY:	visible
MODE:	modal
PARAMETER:	""
BORDER:	none
TITLE BAR:	none
SYSTEM MENU:	none

9.1.2 Attributes for Dialog Boxes

If only **border** is specified, the standard one-pixel wide PM window border will be drawn. If **dialog border** is specified, the standard PM dialog border will be drawn, which is a thick non-sizeable border drawn in the PM title bar color. The user can set this color from within the PM Control Panel.

Modal dialog boxes should specify **dialog border**, and modeless dialog boxes should specify **border**.

For a description of the **title bar** attribute see 10.1.2, "Using the Title Bar Attribute" on page 10-4.

For a description of the **system menu** attribute see 10.1.5, "Using the System Menu Attribute" on page 10-5.

9.1.3 Actions for Dialog Boxes

You can use **add**, **change text**, **clear**, **disable**, and **enable** with dialog boxes. The **add** statement is used to create a dialog box at runtime. The **change text** statement is used to change the text which appears in the title bar of the dialog box (if one exists). The **clear** statement will delete all the children in the dialog box. The **disable** and **enable** statements will disable or enable the dialog box. Users cannot interact with a disabled dialog box or any of its children.

9.1.4 Keyboard and Mouse Use in Dialog Boxes

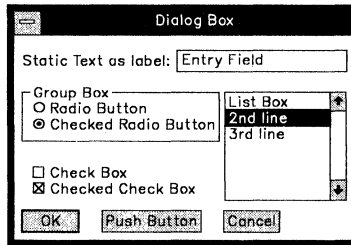
When users are interacting with a dialog box and its controls, they can use the arrow keys to move between elements in a group. Pressing TAB moves from group to group, and to any object not in a group. Pressing ENTER causes the response for the default push button to be taken. Pressing ESCAPE causes the response for the cancel push button to be taken.

The primary mouse selection button can be used to select any object in the dialog box, and make that object the active object.

Pressing the ESCAPE key, double-clicking on the system menu, or selecting the "Close" choice from the system menu invokes a response to the cancel push button, if there is one.

9.1.5 Example of a Dialog Box

The following diagram shows an example of a dialog box that contains a sample of the different dialog control objects. See 9.4, "Sample Code for Generating a Dialog Box" on page 9-24 for the EASEL OS/2 EE code that generated this diagram.



9.1.6 Built-in Functions for Dialog Boxes

The following built-in functions can be used with dialog boxes:

ancestry
ancestry of
exists
object
parameter [of]
selectability of

text of
visibility of
xposition [of]
yposition [of]
xsize [of]
ysize [of]

9.2 Dialog Regions

A dialog region is a rectangular portion of the screen into which dialog control objects can be placed. Dialog regions can have any combination of dialog control objects (see 9.3, “Dialog Control Objects” on page 9-7).

A dialog region can have only dialog controls as its children.

The initial attributes of a dialog region are defined in the **dialog region** statement. You can change any of these attributes during runtime. The definition uses the following model.

Model

```
[ enabled | disabled ] [ visible | invisible ]
[primary] dialog region DR_NAME
  size XSIZE YSIZE
  at { [position] X Y | default position }
  [in PARENT_NAME1 [in PARENT_NAME2]... ]
  [priority is PRIORITY_NO]
  [parameter is PAR_STRING]

[REGION_ATTRIBUTES]
```

The REGION_ATTRIBUTES that you can use to define the appearance of the dialog region are presented in 10.1, “Region Attributes” on page 10-2.

Any of the attributes after ancestry can be specified in any order.

Only the **dialog region** specification, the identifier, the **at [position]** specification, and the **size** specifications are required. Other attributes are optional, using the defaults shown below.

Default Values

SELECTABILITY:	enabled
VISIBILITY:	visible
ANCESTRY (primary):	in desktop
ANCESTRY:	in primary region (in master for fullscreen)
PRIORITY:	0
PARAMETER:	""

A dialog region can be used wherever a graphical, image, or textual region can be used, and it can have any of the window frame attributes (title bar, action bar, etc.) that can be used in these regions.

The coordinates used in the definition of a dialog region (and therefore its children) are dialog units.

9.2.1 Actions for Dialog Regions

You can use **add**, **change text**, **clear**, **disable**, and **enable** with dialog regions. The **add** statement is used to create a dialog region at runtime. The **change text** statement is used to change the text which appears in the title bar of the dialog region (if one exists). The **clear** statement will delete all the children in the dialog region. The **disable** and **enable** statements will disable or enable the dialog region. Users cannot interact with a disabled dialog region or any of its children.

9.2.2 Keyboard and Mouse Use in Dialog Regions

When users are interacting with a dialog region and its children, they can use the arrow keys to move between elements in a group. Pressing TAB moves from group to group, and to any object not in a group. Pressing ENTER causes the response for the default push button to be taken, if one exists. Pressing ESCAPE causes the response for the cancel push button to be taken, if one exists. The keyboard can also be used to interact with the dialog action bar, if one exists.

The primary mouse selection button can be used to select any object in the dialog region, and make that object the active object.

Pressing the ESCAPE key, double-clicking on the system menu, or selecting the "Close" choice from the system menu invokes a response to the cancel push button, if there is one. Otherwise the **response to ... on close** is invoked.

9.2.3 Built-in Functions for Dialog Regions

The following built-in functions can be used with dialog boxes:

ancestry	selectability of
ancestry of	text of
bottom [of]	top [of]
exists	visibility of
left [of]	xposition [of]
object	yposition [of]
parameter [of]	xsize [of]
right [of]	ysize [of]

9.3 Dialog Control Objects

9.3.1 Overview

The dialog control objects, which can be children of dialog boxes or dialog regions, are as follows:

- entry field** (single-line)
- multiline entry field**
- list box**
- combination box** (simple and drop-down)
- dropdown list**
- push button**
- radio button**
- check box**
- static text**
- group box**

Each dialog control must have a dialog box or dialog region as its parent.

The size and position of dialog controls are defined in dialog units.

When you create dialog boxes or dialog regions, take care not to overlap dialog control objects.

9.3.2 Groups

All controls, except static text objects and group box objects, can be put into groups when they are defined. Groups are used to indicate that a set of items logically belong together. Only objects of the same type should be put into the same group. If the group name referenced has not previously been referenced, then a new group will be created. If the group exists, then this object will be added to the group.

9.3.3 Mnemonics

When you create text for push buttons, radio buttons, and check boxes, you can specify a mnemonic for them. The mnemonic provides a way to select the objects with the keyboard without having to first move to the object to make it active. You define the mnemonic by putting a “~” character before the letter in the text string you want to be the mnemonic. When the text is displayed, this letter is underlined to indicate to the user that it is the mnemonic. Whenever the dialog box or dialog region is active, the user can type this letter in combination with the ALT key to select the object.

9.3.4 Built-in Functions for Dialog Controls

The following built-in functions can be used with dialog controls:

ancestry	text of
ancestry of	visibility of
exists	xposition [of]
object	yposition [of]
parameter [of]	xsize [of]
selectability of	ysize [of]

9.3.5 Single-Line Entry Fields

A single-line entry field consists of a rectangular box into which users type textual information limited to one line. When a single line entry field is the active object, all text typed from the keyboard is automatically placed into it. The following is the model for a single-line entry field.

Model

```
[ enabled | disabled ] [ visible | invisible ]
[password] entry field EF_NAME
  size WIDTH HEIGHT
  at [position] X Y
  in DB_NAME
  [parameter is PAR_STRING]
  [text size TEXT_LIMIT columns]
  [ { left | right | [horizontal] center } align]
  [group is GROUP_NAME]
  [text TEXT_STRING]
```

Default Values

SELECTABILITY:	enabled
VISIBILITY:	visible
PARAMETER:	""
TEXT SIZE:	32
ALIGNMENT:	left
TEXT:	""
GROUP:	none

You can use the **password** keyword to specify that the contents of the entry field will not be visible.

Within the definition of the entry field, you can use the **text size** attribute to specify a limit on the number of characters allowed in the field, for example:

```
text size 15 columns
```

specifies that you can enter only 15 characters in the entry field.

You can also specify an alignment type of right, left, or horizontal center for entry fields, for example:

```
right align
```

specifies that text entered into the entry field will be right aligned.

The **text** attribute lets you specify an initial value for the entry field. The **text of** built-in function returns the current value that the user enters.

9.3.5.1 Actions for Entry Fields

The following action statements can be used with **entry field** objects: **add**, **change text**, **clear**, **disable**, and **enable**. The **add** statement creates an entry field object at runtime. The **change text** statement changes the text displayed in the entry field. The **clear** statement clears out all text currently in the entry field. The **disable** statement prevents the user from moving to or modifying the entry field. The **enable** statement causes the entry field to be drawn in normal color and allows the user to move to and/or modify the entry field.

9.3.5.2 Entry Field Responses

A response will be taken for an entry field every time the contents of the entry field are changed.

9.3.6 Multiple-Line Entry Fields

A multiple-line entry field consists of a rectangular box into which users type information that consists of more than one line. When a multiple-line entry field is the active object, all text typed from the keyboard is automatically placed into it. The following is the model for a multiple-line entry field.

```
[ enabled | disabled ] [ visible | invisible ] [ [color] COLOR]
multiline entry field EF_NAME
    size WIDTH HEIGHT
    at [position] X Y
    in DB_NAME
    [ [color] COLOR foreground]
    [parameter is PAR_STRING]
    [ignore tab]
    [word wrap]
    [ text { width XLIM | height YLIM } |
      text width XLIM height YLIM ]
    [text size TEXT_LIMIT]
    [horizontal scroll bar]
    [vertical scroll bar]
    [group is GROUP_NAME]
    [font FONT_NAME]
    [text TEXT_STRING]
```

Default Values

SELECTABILITY:	enabled
VISIBILITY:	visible
COLOR (background):	window background (color 26)
COLOR (foreground):	window text (color 27)
PARAMETER:	""
TEXT:	""
FONT:	system

Within the definition of the multiple-line entry field, you can specify the color of the window background, and the color and font of the text. For example:

```
blue multiline entry field MessageText
:
white foreground
font "asc713"
```

If you specify **ignore tab**, the multiple-line entry field will ignore the TAB key. This will allow the use of the TAB key to move to the next control in the dialog box, and is required by CUA in a dialog box.

The **word wrap** attribute specifies that the system will automatically move the cursor to the next line when the maximum text width is reached. The maximum text width is the end of the visible field if no **text width** is specified. Otherwise, the maximum text width is the value specified in the **text width** attribute. Note that spaces are allowed to extend past the maximum text width.

You can use the **text width ... height ...** attributes to specify the maximum extent of the text entered into the multiple-line entry field, expressed in pixels. By default, the maximum text extent is defined by the window size if the multiple-line entry field has no scroll bar. Otherwise, the text extent is unlimited. Note that if **word wrap** is specified, spaces are allowed to extend past the maximum width.

Use the **text size** attribute to specify a limit on the number of characters allowed in the field.

The **text** attribute lets you specify an initial value for the multiple-line entry field. The text can contain line-feed characters ("`\n`") or carriage return and line-feed pairs ("`\r\n`") to denote a line break. If you have specified **word wrap**, the text will be formatted to fit within the given text width. If you have specified any text limits or have not specified scroll bars, make sure the initial text will fit within the limits.

The **text of** built-in function returns the current value that the user enters. The text will contain a carriage return and line-feed pair ("`\r\n`") where the user has entered a return, denoting a hard line break. Soft line breaks (produced by word wrap rather than by the use of the RETURN key) are not denoted in the string.

9.3.6.1 Actions for Multiple-Line Entry Fields

The following action statements can be used with multiple-line entry field objects: **add**, **change text**, **clear**, **disable**, and **enable**. The **add** statement creates a multiple-line entry field object at runtime. The **change text** statement changes the text displayed in the multiple-line entry field. The **clear** statement clears out all text currently in the multiple-line entry field. The **disable** statement prevents the user from moving to or modifying the multiple-line entry field. The **enable** statement causes the multiple-line entry field to be drawn in normal color and allows the user to move to and/or modify the entry field.

9.3.6.2 Multiple-Line Entry-Field Responses

A response will be taken for a multiple-line entry field every time the contents of the entry field are changed.

9.3.7 List Boxes

List boxes display a list of choices from which the user can select. A list box should be wide enough to display the longest line of text that it will contain. If the list of choices is longer than can be displayed, the list can be scrolled. When a list box is active, the user moves through it with the arrow keys. For a single selection list box, the arrow keys are used to change the current selection. For a multiple selection list box, the spacebar or left

mouse button are used to select or deselect one or more lines. The model for a list box is:

Model

```
[ enabled | disabled ] [ visible | invisible ]
[multiple selection] list box LB_NAME
    size WIDTH HEIGHT
    at [position] X Y
    in DB_NAME
    [parameter is PAR_STRING]
    [sort { ascending | descending } ]
    [horizontal scroll bar]
    [group is GROUP_NAME]
    [insert { TEXT_STRING | file ASCII_FILE_NAME } ]...
```

Default Values

SELECTABILITY:	enabled
VISIBILITY:	visible
MULTIPLE SELECTION:	no
PARAMETER:	""
GROUP:	none

List boxes allow you to define how the entries in the list box should be displayed. The default is not to sort the list at all, in which case each new entry into the list box will just go at the bottom. You can also specify a sorting order of ascending or descending, for example:

sort descending

causes the entries in the list box to be sorted in descending order. To add entries into a list box, the insert statement is used. You can either insert strings or insert a file. When inserting strings, each new insert statement will start a new entry in the list box. If a string contains the character “\n” then the text following the “\n” will start a new entry. When inserting a file, each line of the file will be used as a new list box entry.

9.3.7.1 Actions for List Boxes

The following action statements can be used with **list box** objects: **add**, **add to**, **change window position**, **clear**, **disable**, **enable**, **for each selected line**, **read**, **remove**, **select**, and **deselect**. The **add** statement creates a list box object at runtime. The **add to** statement allows you to insert options into the list box at runtime. The **change window position** statement allows you to indicate which block of lines should be displayed. The **clear** statement clears out all entries in the list box. The **disable** statement prevents the user from moving to or selecting from the list box. The **enable** statement

causes the list box to be drawn in normal color and allows the user to move to and/or select from the list box.

The **for each selected line** statement allows you to perform a set of actions for each selected line in a multiple selection list box.

Model

```
for each selected line of LB_NAME loop [LOOP_NAME]
```

```
    [ACTION_STATEMENT]...  
    [leave loop [LOOP_NAME] ]
```

```
end loop
```

When in a **for each selected line** statement, the special inquiry built-in function **selected line** returns the line number of the currently selected line within the list box.

Do not use **read** or **insert** to change the contents of a list box while executing this loop. You can use **remove** to remove one or more lines.

The **read** statement is used to insert the lines of the specified file into the list box. This has the same effect as the **insert file** statement except that the **read** statement is executed at runtime.

Model

```
read file ASCII_FILE_NAME into LB_NAME
```

The **remove** statement is used to remove one or more lines from a list box.

Model

```
remove [line] LINE_NO [thru [line] LINE_NO] from LB_NAME
```

The **select** and **deselect** statements are used to select or deselect a given line (a selected line appears in reverse video.) If the list box is a single selection list box, the previously selected line is deselected when a selection is made.

Model

```
{ select | deselect } line LINE_NO in LB_NAME
```

9.3.7.2 Built-in Functions for List Boxes

A list box can also be used in conjunction with the built-in functions: **bottom [of]**, **textual line from**, **top [of]**, **selected line**, **selected line from**, and **window yposition of**. The **bottom [of]** function returns the line number of the last line in the list box. The **top [of]** function returns the line number of the first line (which is always 1). The **textual line from** function can be used to get the text that appears on the specified line of the list box.

Model

```
textual line LINE_NO from LB_NAME
```

The **selected line** function is used to identify the currently selected line in a **for each selected line** loop.

Model

```
selected line
```

The **selected line from** function is used to get the line number of the currently selected line in a single select list box. If used on a multiple select list box, it returns only the first line selected.

Model

```
selected line from LB_NAME
```

The **window yposition of** function is used to determine the line under the first line within the list box.

9.3.7.3 List Box Responses

A response for a list box is taken whenever a selection or deselection is made. This happens whenever the user selects an item with the mouse, or in single selection list boxes when the user moves the arrow keys or in multiple selection list boxes when the user presses the space bar.

Note that for multiple selection list boxes, if an entry is previously selected, pressing the space bar, single-clicking or double-clicking on the entry again will deselect that entry.

9.3.8 Combination Boxes

A combination box combines the capabilities of both a single line entry field and a list box. The user can enter a value into the entry field by selecting a line from the list box, or by typing a value directly into the entry field. The following is the model for a combination box.

Model

```
[ enabled | disabled ] [ visible | invisible ]
[dropdown] combination box CB_NAME
    size WIDTH HEIGHT
    at [position] X Y
    in DB_NAME
    [parameter is PAR_STRING]
    [text size TEXT_LIMIT columns]
    [sort { ascending | descending } ]
    [group is GROUP_NAME]
    [text EF_STRING]
    [insert { LB_STRING | file ASCII_FILE_NAME } ]...
```

Default Values

SELECTABILITY:	enabled
VISIBILITY:	visible
PARAMETER:	""
TEXT SIZE:	32
TEXT:	""

9.3.8.1 Drop-down Combination Box

The keyword **dropdown** specifies that this is a drop-down combination box. In this case, the list box is normally hidden, and a down arrow is displayed to the right of the entry field. Selecting the arrow displays the list box, allowing the user to make a selection. The list box is then hidden.

The size and position of the combination box include both the entry field and list box, even if the list box is hidden.

9.3.8.2 Actions and Built-in Functions

The actions and built-in functions available for a single selection list box control are also available for a combination box. For example:

- The **selected line from** function returns the line number of the item that the user has selected from the list box.
- The **text of** function returns the text displayed in the entry field.
- The **change text** statement changes the text displayed in the entry field.
- The **clear** statement clears the contents of both the list box and the entry field.

The actions available for a single-line entry field control are also available for a combination box.

9.3.9 Drop-down Lists

A drop-down list has a selection field with one choice displayed as the default value, and a hidden list box. When the user selects the down arrow displayed to the right of the field, the list box appears. The user can then select a line from the list box to change the choice displayed. The following is the model for a drop-down list.

Model

```
[ enabled | disabled ] [ visible | invisible ]
dropdown list DL_NAME
  size WIDTH HEIGHT
  at [position] X Y
  in DB_NAME
  [parameter is PAR_STRING]
  [text size TEXT_LIMIT columns]
  [sort { ascending | descending } ]
  [group is GROUP_NAME]
  [text FIELD_STRING]
  [insert { LB_STRING | file ASCII_FILE_NAME } ]...
```

Default Values

SELECTABILITY:	enabled
VISIBILITY:	visible
PARAMETER:	""
TEXT SIZE:	32
TEXT:	""

The size and position of the drop-down list include both the selection field and list box.

9.3.9.1 Actions and Built-in Functions

The actions and built-in functions available for a single selection list box control are also available for a drop-down list. For example:

- The **selected line from** function returns the line number of the item that the user has selected from the list box.
- The **text of** function returns the text displayed in the selection field.
- The **change text** statement changes the text displayed in the selection field.
- The **clear** statement clears the contents of both the list box and selection field.

The actions available for a single-line entry field control are also available for a dropdown list.

9.3.10 Push Buttons

Push buttons allow the user to perform an immediate action by selecting the button. Push buttons usually appear at the bottom of a dialog box and are used for functions such as OK and CANCEL. The following is the model for a push button:

Model

```
[ enabled | disabled ] [ visible | invisible ]
[cancel] [default] push button PB_NAME
    size WIDTH HEIGHT
    at [position] X Y
    in DB_NAME
    [parameter is PAR_STRING]
    [no border]
    [group is GROUP_NAME]
    [text TEXT_STRING]
```

Default Values

SELECTABILITY:	enabled
VISIBILITY:	visible
PARAMETER:	""
CANCEL:	no
DEFAULT:	no
BORDER:	visible
TEXT:	""

A push button with the **default** attribute has its response taken if the user presses ENTER. A push button with the **cancel** attribute has its response taken if the user presses the ESCAPE key, selects the "Close" choice from the parent's system menu, or double-clicks on the parent's system menu. Specify the **no border** attribute if you do not want a rectangular border. Use the **text** attribute to specify the text to appear in the button.

9.3.10.1 Actions for Push Buttons

The following action statements can be used with **push button** objects: **add**, **change text**, **clear**, **disable**, and **enable**. The **add** statement creates a push button object at runtime. The **change text** statement changes the text displayed in the push button. The **clear** statement clears out all text currently in the push button. The **disable** statement causes the push button to be grayed and prevents the user from moving to or selecting the push button. The **enable** statement causes the push button to be drawn in normal color and allows the user to move to and/or select the push button.

9.3.10.2 Push Button Responses

A response for a push button is taken whenever a selection is made. This happens whenever the user selects the button with the mouse, or selects the space bar when the button is the active object. The **default** push button has its response taken also when the user presses RETURN while in the dialog box and the **cancel** push button has its response taken when the user presses ESCAPE while in the dialog box, selects the "Close" choice from the parent's system menu, or double-clicks on the parent's system menu.

9.3.11 Radio Buttons

Radio buttons appear as hollow or filled circles with text beside the circles. They are usually put together into a group. If grouped, only one radio button in the group can be selected at a time (the selected radio button has its circle filled). When a user selects a radio button, PM fills that button and clears the others in its group. You can write a response for an individual radio button. However, you will usually wait until the dialog box is finished with and then use the built-in function **checked in group** to find out which selection was the final one. The following is the model for a radio button.

Model

```
[ enabled | disabled ] [ visible | invisible ] [checked]
radio button RB_NAME
  size WIDTH HEIGHT
  at [position] X Y
  in DB_NAME
  [parameter is PAR_STRING]
  [group is GROUP_NAME]
  [text TEXT_STRING]
```

Default Values

SELECTABILITY:	enabled
VISIBILITY:	visible
PARAMETER:	""
CHECKED:	unchecked
GROUP:	none
TEXT:	""

If a radio button is specified with the attribute of **checked**, its circle is drawn as filled when the button is first displayed. Use the **text** attribute to specify the text to display beside the circle. The radio button should be big enough to display the specified text.

9.3.11.1 Actions for Radio Buttons

The following action statements can be used with **radio button** objects: **add**, **change text**, **clear**, **disable**, **enable**, **check**, and **uncheck**. The **add** statement creates a radio button object at runtime. The **change text** statement changes the text displayed after the circle in the radio button. The **clear** statement clears out the text in the radio button. The **disable** statement causes the radio button to be grayed and prevents the user from moving to or selecting the radio button. The **enable** statement causes the radio button to be drawn in normal color and allows the user to move to and/or select the radio button. The **check** statement allows you to set the state of the radio button as selected (fill in its circle). The **uncheck** statement allows you to set the state of the radio button as deselected (hollow out its circle).

Model

```
{ check | uncheck } OBJECT_NAME
```

9.3.11.2 Built-in Functions for Radio Buttons

You can also use the following built-in functions with radio buttons: **is checked** and **checked in group**. Use the **is checked** function to query the state of a given radio button. It returns **true** if the radio button is selected.

Model

OBJECT_NAME is checked

The **checked in group** function returns the name of the selected radio button in a group of radio buttons from a particular dialog box or dialog region.

Model

checked in group RBG_NAME from DB_NAME

9.3.11.3 Radio Button Responses

A response for a radio button is taken whenever it is selected, that is, whenever the user selects the button with the mouse, or in keyboard mode, whenever the radio button is moved to with the arrow keys.

9.3.12 Check Boxes

Check boxes appear as hollow or checked boxes with text beside the boxes. They are usually put together into a group. When a user selects a check box, PM toggles state of the box: if the box was checked, PM clears it; if it was unchecked, PM checks it. You can have many check boxes within a group checked, unlike radio buttons. Use the built-in function **is checked** to query the state of a given check box. The following is the model for a check box.

Model

[enabled | disabled] [visible | invisible] [checked]
check box CB_NAME
 size WIDTH HEIGHT
 at [position] X Y
 in DB_NAME
 [parameter is PAR_STRING]
 [group is GROUP_NAME]
 [text TEXT_STRING]

Default Values

SELECTABILITY:	enabled
VISIBILITY:	visible
PARAMETER:	""
CHECKED:	unchecked
TEXT:	""
GROUP:	none

If a check box is specified with the attribute of **checked**, its box is drawn as checked when the button is first displayed. Use the **text** attribute to specify the text to be displayed beside the box.

9.3.12.1 Actions for Check Boxes

The following action statements can be used with **check box** objects: **add**, **change text**, **clear**, **disable**, **enable**, **check**, and **uncheck**. The **add** statement creates a check box object at runtime. The **change text** statement changes the text displayed after the box. The **clear** statement clears out the text. The **disable** statement causes the check box to be grayed and prevents the user from moving to or selecting the check box. The **enable** statement causes the check box to be drawn in normal color and allows the user to move to and/or select the check box. The **check** statement allows you to set the state of a check box as selected (fill its box). The **uncheck** statement allows you to set the state of the check box as deselected (hollow out its box).

9.3.12.2 Built-in Function for Check Boxes

You can also use the **is checked** built-in function with check boxes. The **is checked** function is used to query the state of a given check box. It returns **true** if the check box has been selected.

9.3.12.3 Check Box Responses

A response for a check box is taken whenever it is selected or deselected, that is, whenever the user selects the check box with the mouse, or in keyboard mode, when the user selects the space bar when the check box is the active object.

9.3.13 Static Text

Static text is used for information purposes. When defining static text you define a box size, which is used to determine boundary information for the text, a string, and some formatting information. The following is the model for a static text object.

```
[ enabled | disabled ] [ visible | invisible ]
static text ST_NAME
  size WIDTH HEIGHT
  at [position] X Y
  in DB_NAME
  [parameter is PAR_STRING]
  [ { left | right | [horizontal] center } [align]
  { top | bottom | vertical center } align]
  [word wrap]
  [text TEXT_STRING]
```

Default Values

```
SELECTABILITY:  enabled
VISIBILITY:     visible
PARAMETER:      ""
ALIGNMENT:      top left
WORD WRAP:      no
TEXT:           ""
```

The initial value of the text in a static text object is specified with the **text** attribute. When you specify the **word wrap** attribute, the specified text is wrapped into as many lines as will fit in the size box specified. If more text than will fit is specified, the remaining text is clipped. You can also specify an alignment type to specify how the text will appear within the box. For example:

```
right vertical center align
```

causes the text to be right aligned and also vertically centered within the bounds of the box specified. Static text that is bottom aligned and word wrapped will display only the first line of text.

9.3.13.1 Actions for Static Text Objects

The following action statements can be used with **static text** objects: **add**, **change text**, **clear**, **disable**, and **enable**. The **add** statement creates a static text object at runtime. The **change text** statement changes the text displayed in the static text object. The **clear** statement clears out all text currently in the static text object. The **disable** statement causes the static text to be grayed. The **enable** statement causes the static text to be drawn in normal color.

9.3.14 Group Boxes

Group box objects are used for information purposes. A group box is displayed as a rectangle with an optional text string that appears at the top of the box, starting on the left. The following is the model for a group box.

Model

```
[ enabled | disabled ] [ visible | invisible ]
group box GB_NAME
    size WIDTH HEIGHT
    at [position] X Y
    in DB_NAME
    [parameter is PAR_STRING]
    [text TEXT_STRING]
```

Default Values

SELECTABILITY:	enabled
VISIBILITY:	visible
PARAMETER:	""
TEXT	""

Group boxes are usually used to delineate which dialog control objects are in a given group or set of groups.

9.3.14.1 Actions for Group Boxes

The following action statements can be used with **group box** objects: **add**, **change text**, **clear**, **disable**, and **enable**. The **add** statement creates a group box object at runtime. The **change text** statement changes the text displayed in the top of the box. The **clear** statement clears out all text currently in the group box. (This does not affect any objects displayed within the box; only the text drawn on the top of the box.) The **disable** and **enable** statements are allowed, but have no effect on the group box.

9.4 Sample Code for Generating a Dialog Box

This code generates the dialog box shown in 9.1.5, "Example of a Dialog Box" on page 9-4.

```
#
# Dialog Box Object Definition(s)
#

enabled dialog box DialogBox1
    size 200 110
    at position 50 40
    dialog border
    title bar "Dialog Box"
    system menu

enabled visible static text Control5
    size 90 8
    at position 6 92
    in DialogBox1
    left align
    top align
    word wrap
    text "Static Text as label:"

enabled visible entry field Control6
    size 102 11
    at position 98 90
    in DialogBox1
    text size 30 columns
    left align
    text "Entry Field"

disabled visible group box Control11
    size 116 33
    at position 4 52
    in DialogBox1
    text "Group Box"

enabled visible multiple selection list box
Control8
    size 80 64
    at position 122 17
    in DialogBox1
    insert "List Box"
    insert "2nd line"
    insert "3rd line"
```



```
enabled visible radio button Control2
    size 78 11
    at position 7 66
    in DialogBox1
    group is SelectionField
    text "Radio Button"

enabled visible checked radio button Control3
    size 110 11
    at position 7 55
    in DialogBox1
    group is SelectionField
    text "Checked Radio Button"

enabled visible check box Control4
    size 64 10
    at position 7 37
    in DialogBox1
    text "Check Box"

enabled visible checked check box Control7
    size 104 10
    at position 7 28
    in DialogBox1
    text "Checked Check Box"

enabled visible default push button OK
    size 38 12
    at position 12 4
    in DialogBox1
    group is Actions
    text "~OK"

enabled visible cancel push button Cancel
    size 38 12
    at position 156 3
    in DialogBox1
    group is Actions
    text "Cancel"

enabled visible push button Control9
    size 76 12
    at position 68 3
    in DialogBox1
    text "Push Button"
```

Chapter 10. Region Attributes and Action Bars

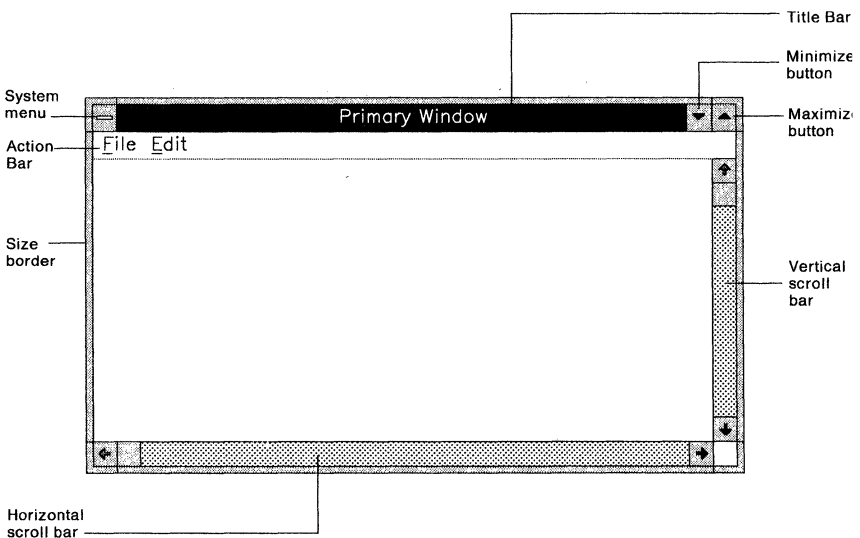
10.1 Region Attributes

This chapter describes the following region attributes:

- Border
- Title bar
- Minimize/Maximize buttons
- System menu
- Scroll bars
- Action bar
- No scale
- Pointer

You can apply these attributes to all types of regions except sense regions. The first six attributes listed, referred to as *frame attributes*, are standard Presentation Manager (PM) user interface components that allow the user to interact with the window. PM controls user interaction with the frame attributes independently of EASEL OS/2 EE.

The following diagram illustrates the frame attributes.



The following model gives the EASEL OS/2 EE syntax for defining region attributes.

Model

[NORMAL_REGION_DEFINITION]

```
[ [ size | [color] COLOR ] border]
[title bar TITLE_STRING]
[minimize button [using MINIMIZE_ICON_FILE] ]
[maximize button]
[system menu]
[horizontal scroll bar [scroll by SCROLL_INC] ]
[vertical scroll bar [scroll by SCROLL_INC] ]
[action bar AB_NAME]
[no scale]
[pointer [is] POINTER_FILE]
```

In the model above, NORMAL_REGION_DEFINITION is the definition of a graphical, textual, image, or dialog region. Any or all of the attributes can be specified in the region definition, in any order, with the following exceptions:

- **textual region:** always no scale.
- **image region:** uses scaling specification shown in 7.5.3, “Defining Image Regions” on page 7-64.
- **dialog region:** always no scale; cannot specify border color; cannot control pointer appearance.

Note: None of these attributes can be changed during runtime except for the following:

- Border color and border visibility of graphical, image, and textual regions, if the region does not have any other frame attributes.
- Title bar text string.
- Action bar and action bar contents.

10.1.1 Using the Border Attribute

The border attribute allows you to specify whether a region should have a one pixel wide or size border around the perimeter of the region's viewport. If the region has no other frame attributes, and a line border is desired, you can also specify what color it should be drawn in:

Model

[[color] COLOR] border

If a line border is desired and other frame attributes are being specified for the region, the color of the line will be whatever color has been set in PM for the window border color. The user can set this color from within the PM Control Panel.

A border color cannot be specified for a graphical, image, or textual region that has an ancestry of **in desktop**. If the region also does not have any other frame attributes, **border** cannot be specified either.

If a size border is desired, it will be the color of the PM's border color. The user can set its color and width from within the PM Control Panel. A size border on the region allows the user to resize the region. PM handles the actual resizing of the region. This means that the EASEL OS/2 EE developer does not have to write a response for the object in order to allow the user to resize it.

10.1.2 Using the Title Bar Attribute

The title bar attribute allows you to specify a title for a region. The title `TITLE_STRING` is centered in a bar, which is drawn above the viewport. The color of the text and bar are PM colors and are set in the PM Control Panel.

In addition to labeling a region, title bars are used by PM to allow the user to move a region around the screen. When users click on the title bar, PM displays a surrounding box that they can move around the screen until they release the mouse button. PM then moves the region to the new position. PM controls the whole interaction. This means that the EASEL OS/2 EE developer does not have to write a response for the object in order to allow the user to reposition it.

10.1.3 Using the Minimize Attribute

The minimize attribute allows you specify a minimize button for the region. PM draws the minimize button in the upper right corner above the viewport. The user can set the color in the PM Control Panel.

When the user clicks the minimize button, the region is made invisible and the icon specified in the `MINIMIZE_ICON_FILE` is drawn on the screen. If you do not specify an icon file, the standard PM application icon (which looks like a blank screen) is used. Double clicking on the icon will make the icon invisible and the region visible.

10.1.4 Using the Maximize Attribute

The maximize attribute allows you specify a maximize button for the region. PM draws the maximize button in the upper right corner above the viewport. If you also defined a minimize button, it is moved to the left of the maximize button. The user can set the color in the PM Control Panel.

When the user clicks the maximize button, the region is enlarged to be fullscreen, and a restore button is displayed in place of the maximize button. When the user selects the restore button, the region is redrawn at the size and position it was at before the maximize button was selected.

10.1.5 Using the System Menu Attribute

The system menu attribute specifies that the region will contain a system menu. The system menu button will be in the upper left corner of the region above the viewport and to the left of the title bar (if there is one). When the user clicks on the system menu button the following options are displayed:

- Restore**
- Move**
- Size**
- Minimize**
- Maximize**
- Close**
- Task Manager**

If the region is minimized or maximized, the user can select Restore to restore the region to its size and position before the minimize or maximize was invoked.

The user can select Move to move the region. (Clicking on the title bar also allows you to move the region.)

If a size border has been specified for the region, the user can select Size to resize the region. (You can also use the size border to resize the region.)

If a minimize button has been defined for the region, the user can select Minimize to minimize the region, unless the region is already minimized.

If a maximize button has been defined for the region, the user can select Maximize to maximize the region, unless the region is already maximized.

The user can select Close to close the region. Double clicking on the system menu button also closes the region.

The user can select Task Manager to bring up the task manager window and switch to a task other than the current session of EASEL OS/2 EE.

If there is no **on close** response to invoke for a graphical, image, or textual region, double-clicking on the system menu or selecting the “Close” choice from the system menu destroys the region. If there is no cancel pushbutton response and **on close** response to invoke for a dialog region, double-clicking on the system menu or selecting the “Close” choice from the system menu destroys the region.

CAUTION:

Be careful when you define the primary region to have a system menu. If the user closes the primary region using the system menu, the EASEL OS/2 EE session stops and the EASEL OS/2 EE program cannot respond to the exit, unless a response to primary region on close statement exists.

10.1.6 Using the Scroll Attributes

The scroll attributes allow you to specify that a region will have a horizontal and/or vertical scroll bars associated with it. The vertical scroll bar is displayed on the outer right edge of the region's viewport. The horizontal scroll bar is displayed below the region's viewport. The user can set the scroll arrow color in the PM Control Panel.

If there is nothing outside the region's window, the scroll bars for the region are displayed as hollow. If there is information outside the region's window, a slider is drawn inside the scroll bar, showing the user where in the region the window is, compared with the available information. The user can either select the scroll arrows or click on the slider and drag it to a new position to change the window position.

When the arrow is selected, the region is scrolled by the number of window coordinates specified by SCROLL_INC. In order to move a full page, the user can select the area between the slider and the scroll arrows. The page increment used is the X or Y window size minus one scroll increment. The user can use the PM Control Panel to change the color of the scroll bars.

10.1.7 Using the Action Bar Attribute

The action bar attribute allows you to give the region an action bar that is created from the action bar template AB_NAME. The user can set the action bar color in the PM Control Panel. (See 10.3, “Creating Action Bars” on page 10-9 for more information on action bar templates and their usage.)

10.1.8 Using the No Scale Attribute

The no scale attribute allows you to specify how the region will scale when it is resized, either by the user via the size border or by statements within the EASEL OS/2 EE program. If you specify no scale, the image within the region's viewport is not scaled when the user resizes the region with size border. Instead, more or less of the region is revealed. The default is to scale the contents of the region so that the same image is always visible in the window, no matter what size it is. It is best to specify no scale when the contents of the region are mainly textual.

Specifying the **no scale** attribute also determines the ratio between window coordinates and screen pixels. When **no scale** is specified, EASEL OS/2 EE constrains the window size of the region so that one window coordinate is equivalent to one screen pixel.

10.1.9 Using the Pointer Attribute

The pointer attribute allows you to specify the appearance of the mouse pointer when it is over the region. The pointer used is found in the file `POINTER_FILE`, which should contain the bitmap pointer. The pointer file should have the format output by the OS/2 icon editor.

10.2 Region Actions

The **change text** action is used to change the text which appears in the title bar.

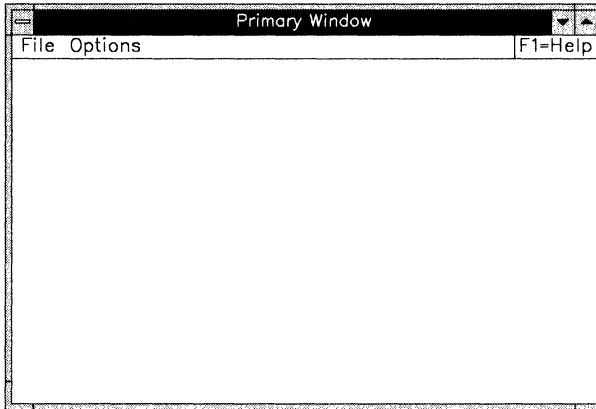
Model

change OBJECT_NAME **text to** TEXT_STRING

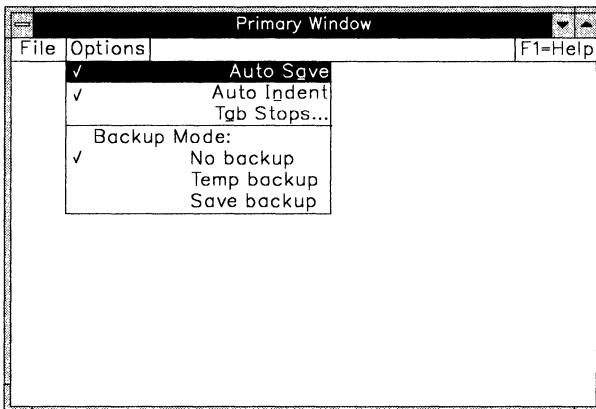
When this action is executed, the text in the title bar of OBJECT_NAME is updated to be the text, TEXT_STRING. If a title bar is not specified for OBJECT_NAME, this action has no effect.

10.3 Creating Action Bars

An action bar is a list of options displayed horizontally above a region's window and below the title bar (if one is defined). An action bar is made up of items called *pulldowns* and buttons.



Buttons appear at the right of the action bar. They are separated from the pulldowns by a vertical bar. **Note:** Buttons do not comply with CUA guidelines. The items in the left portion of the action bar are called *pulldowns*. When the text of a pulldown is selected with a mouse, the pulldown menu appears.



As the diagram shows, pulldowns also contain items, displayed vertically. The textual pulldown items are called choices, and the horizontal bars are called separators.

Responses can be written for each button in an action bar and each choice in its pulldowns. Buttons and choices are selected by clicking on them with the mouse. In addition, keyboard shortcuts can be defined in the form of accelerator keys and mnemonics, described in 10.3.7, “Accelerator Keys” on page 10-14 and 10.3.8, “Mnemonics” on page 10-15.

The appearance of a region’s action bar can be changed at runtime by replacing it with a new action bar or manipulating its items.

Pulldowns, choices, separators and buttons are all considered items of an action bar. Items in a region’s action bar can be added or deleted dynamically, or their status can be changed.

10.3.1 Defining an Action Bar Template

An action bar is described by an *action bar template*. This template defines the initial appearance of an action bar in a region. It contains a description of the buttons and pulldowns in the action bar. A single action bar template can be used to create action bars in many different regions.

Once a region’s action bar has been created from a template, its appearance can be changed by adding, deleting and manipulating its items. These changes affect *only* the region’s action bar, *not* the template that was used to create it. Action bar templates cannot be modified after their creation.

Model

```
action bar AB_NAME is
```

```
    [PULLDOWN_DEF_OR_REF]...  
    [BUTTON_DEF_OR_REF]...
```

```
end action bar
```

The following code defines an empty action bar.

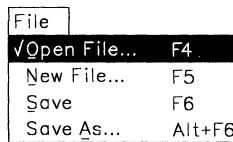
```
action bar MainMenu is  
end action bar
```

This action bar can be included in a region's frame attributes by specifying action bar MainMenu when defining the region.

The following sections describe how to specify pulldowns and buttons in an action bar template.

10.3.2 Defining Pulldowns

Pulldowns are menus which display a vertical list of choices. The text for a pulldown is displayed in the left portion of the action bar. When a pulldown is selected by the user, the menu is "pulled down" (made visible) so that the user can select one of its choices.



Model

```
[ enabled | disabled ] pulldown PD_NAME text PD_TEXT
  [parameter is PAR_STRING]

  [ITEM_DEF_OR_REF]...
```

end pulldown

When you define a pulldown, you define the text that appears in the action bar as well as the list of choices, separators, and pulldowns that it displays. Here is the definition for the Options pulldown.

```
pulldown OptionsMenu text "Options"
  checked choice AutoSave text "\aAuto ~Save"
  checked choice AutoIndent text "\aAuto ~Indent"
    choice TabStops text "\a~Tab Stops..."
  separator
  static choice BackupMode text "Backup Mode:"
  checked choice NoBackup text "\tNo backup"
    choice TempBackup text "\tTemp backup"
    choice SaveBackup text "\tSave backup"
end pulldown
```

Pulldowns can have an associated parameter. The pulldown text, as well as the text of buttons and choices, can contain the mnemonic character, which is described in 10.3.8, “Mnemonics” on page 10-15.

Cascading menus can be created by nesting pulldowns. You can nest a maximum of 25 levels. A pulldown definition may not reference itself, directly or indirectly.

```
pulldown Graph text "~Graph"
  pulldown GraphType text "~Type"
    checked choice BarGraph text "~BarGraph"
    choice PieGraph text "~PieGraph"
    choice LineGraph text "~LineGraph"
  end pulldown
  choice GraphCreate text "~Create"
end pulldown
```

10.3.3 Defining Choices

The sample pulldown in the previous section shows a variety of choice definitions. Choices can be disabled, checked, or static.

If a choice is:	It is:
disabled	grayed and cannot be selected.
checked	displayed with a checkmark to its left.
static	used for informational purposes only and cannot be selected or moved to.

Model

```
[ enabled | disabled ] [checked] [static]
choice CHOICE_NAME text CHOICE_TEXT
  [accelerator is [ control | alt | shift
                  [ control | alt | shift ]... ] ACCEL_KEY]
  [parameter is PAR_STRING]
```

In addition to the mnemonic character, the text can also include the following escape characters for formatting:

<code>\a</code>	right aligns the item text following the escape sequence.
<code>\t</code>	left aligns a second column of text following the escape sequence.

Note: You cannot use `\t` with a static choice.

In the example, the first three choices (**AutoSave**, **AutoIndent** and **TabStops**) are right-aligned; their last character appears in the last column. The final three choices (**NoBackup**, **TempBackup** and **SaveBackup**) are aligned in a

second column of text. Tab stops cannot be set when using `\t` in pulldown choices; PM uses the text strings in all of the choices to find a column in which to line everything up.

Choices can have an associated parameter. In addition, accelerator keys can be defined for selecting choices (see 10.3.7, “Accelerator Keys” on page 10-14).

10.3.4 Defining Separators

A *separator* is a horizontal line drawn between pulldown choices, used to group choices together. It cannot be selected or moved to.

Model

```
separator [SEPARATOR_NAME]
```

If a separator is named when it is defined then it can be referenced later.

10.3.5 Defining Buttons

Up to two buttons can be included in any action bar. The buttons, which appear to the right of the action bar following a vertical line, are typically used to display options that do not belong in a pulldown, such as “F1 = Help”. Buttons are specified after pulldowns in the action bar template.

Note: Buttons do not comply with CUA guidelines.

Model

```
[ enabled | disabled ] button BUTTON_NAME text BUTTON_TEXT
  [accelerator is [ { control | alt | shift }
                  [ control | alt | shift ]... ] ACCEL_KEY]
  [parameter is PAR_STRING]
```

The example below shows how to define a button called HelpButton in an action bar.

```
button HelpButton text "F1 = Help" accelerator is KEY_F1
```

You can respond to this HelpButton by name. The user can select it using either the mouse or the accelerator function key F1. The button text can also contain the mnemonic character, which is described in 10.3.8, “Mnemonics” on page 10-15. Buttons can have an associated parameter.

10.3.6 Defining Global Items

So far, this section has described how to define the action bar items: pulldowns, choices, separators and buttons. When defining an action bar template, a list of pulldowns is specified, followed by a list of buttons. Each pulldown template contains a list of choices and separators. There are two ways to specify each item in an action bar template: *by definition* and *by reference*.

The choices and separators in the pulldown template in 10.3.2, “Defining Pulldowns” on page 10-11 are specified by definition, that is, their definitions appears within the template.

Items can also be included in an action bar template by reference. These items are defined globally—outside the action bar template—and then referenced within the template. Globally defined pulldowns and buttons can be referenced in any number of action bar templates. Globally defined pulldowns, choices, and separators can be referenced in any number of pulldown templates.

item ITEM_NAME

All global items are referenced using the **item** reference keyword. For example, the HelpButton in 10.3.5, “Defining Buttons” on page 10-13, could have been defined outside the action bar template, and then referenced within it:

```
button HelpButton text "F1 = Help"
    accelerator is KEY_F1
action bar MainMenu is
:
    item HelpButton
end action bar
```

Only global items can be referenced in action bar templates this way; items that are defined inside templates cannot be used elsewhere.

10.3.7 Accelerator Keys

Sometimes it is desirable to provide users with a shortcut to select a menu choice. An accelerator key is a key which, when pressed simultaneously with certain control keys, has the same effect as clicking on the choice or button with the mouse. Pressing the accelerator key causes a pulldown choice to be selected even if the pulldown is not currently visible. The accelerator key for a choice or button is specified in the item definition. It consists of a key combined with any or all of the control keys ALT, SHIFT, and CONTROL. When using an accelerator with a control key or keys, all the keys must be held down simultaneously.

The key is either a single character string or an integer constant from the **accel.inc** include file. Be sure to include this file if you want to use special keys such as the function keys.

For example, the button "F1 = Help" provides the function key F1 as an accelerator key to the user:

```
button HelpButton text "F1 = Help"  
    accelerator is KEY_F1
```

Notice that the text of HelpButton tells the user what the accelerator key is. Be sure to identify the accelerator keys in the item's text; the user has no other way of knowing what the item's accelerator may be.

When using a single-character string to define an accelerator key, the character is case-independent. It is the **shift** keyword that determines whether the accelerator key will be uppercase. For example:

```
accelerator is "x"           # accelerator is x  
accelerator is "X"           # accelerator is x  
accelerator is shift "x"      # accelerator is X  
accelerator is shift "X"      # accelerator is X
```

For numbers or special characters, type the actual number or character. For example:

```
accelerator is "4"           # accelerator is 4  
accelerator is "$"           # accelerator is $
```

Note: Be careful to choose a unique accelerator key for each choice or button in an action bar.

10.3.8 Mnemonics

Mnemonics provide another mechanism for using the keyboard to manipulate action bars (and dialog controls, see 9.3.3, "Mnemonics" on page 9-7). The ALT key, pressed alone, begins the keyboard input. Once ALT is released, the user can use the arrow keys to move among the pulldowns in the action bar (and the system menu, if it is present). (You can also use the F10 key to do this if **-fullscreen** is not specified.)

Each time the left or right arrow key is pressed, the text of the next pulldown is highlighted to indicate that it is currently selected. The user can then press ENTER to pull down the currently selected pulldown or system menu.

Now, the arrow keys can be used to move among the choices in the pulldown. The highlighted choices indicate which is current. Pressing ENTER selects the current choice. Pressing ESC returns the pulldown to the action bar. Pressing ESC again exits keyboard mode.

While in keyboard mode, individual choices, buttons and pulldowns can also be selected using their *mnemonic* characters. If the text of an item contains

a tilde (~), the character immediately following the tilde becomes the mnemonic character for that item. Pressing that key while in keyboard mode causes a choice or button to be selected, or a pulldown to be made visible. The mnemonic character is indicated to the user by underlining it in the item's text.



In the File menu pulldown above, the line under the “O” in the Open File choice indicates that “O” is the mnemonic character for that choice. Notice that because both the Save File and Save As choices begin with “S,” the developer chose “A” as the mnemonic character for the Save As choice. The developer is responsible for making each mnemonic character unique when designing the menu.

Below is the definition of the File menu pulldown.

```
checked choice OpenFile text "~Open File..."
  pulldown FileMenu      text "File"
    item OpenFile
      choice NewFile      text "~New File..."
      choice SaveFile     text "~Save"
      choice SaveAs       text "Save ~As..."
  end pulldown
```

Mnemonics can be used only after ALT has been pressed and then released. This contrasts with accelerator keys: if ALT is used in an accelerator key combination, it must be pressed simultaneously with the accelerator key.

10.3.9 Item Parameters

Parameters can be associated with choices, pulldowns and buttons. An item's parameter is associated with the item's definition or template. Even though more than one action bar can be created from the same action bar template, the items in each action bar actually share the parameter from the template.

Note: The parameter of an item in a region's action bar cannot be changed independently of the other items created from the same menu definition.

```
make item { ITEM_NAME [from AB_NAME] | ITEM_CLASS_NAME }  
  parameter PAR_STRING
```

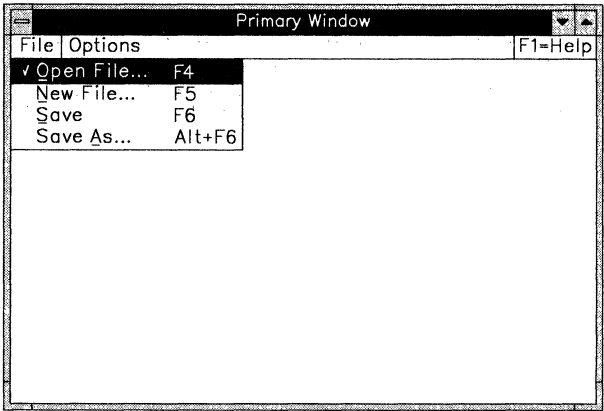
make item parameter always changes the value of the parameter of an item in every region whose action bar contains the item.

10.3.10 An Example Action Bar Template

The following action bar template uses many of the features we have discussed in this section.

```
# This is a pulldown choice defined outside an action-bar template  
checked choice OpenFile text "~Open File...\tF4"  
  accelerator is KEY_F4  
  
# Action Bar Template  
action bar MainMenu is  
  pulldown FileMenu text "File" # This pulldown will be 2 columns  
    item OpenFile # Item reference to predefined choice  
    choice NewFile text "~New File...\tF5"  
      accelerator is KEY_F5  
    choice SaveFile text "~Save\tF6"  
      accelerator is KEY_F6  
    choice SaveAs text "Save ~As...\tAlt+F6"  
      accelerator is alt KEY_F6  
  end pulldown  
  pulldown OptionsMenu text "Options" #This pulldown will be right  
    aligned  
    checked choice AutoSave text "\aAuto ~Save"  
    checked choice AutoIndent text "~\Auto ~Indent"  
    choice TabStops text "\a~Tab Stops..."  
    separator  
    static choice BackupMode text "Backup Mode:" #next 3 items  
    tabbed  
    checked choice NoBackup text "\tNo backup"  
      choice TempBackup text "\tTemp backup"  
      choice SaveBackup text "\tSave backup"  
  end pulldown  
  button HelpButton text "F1 = Help" accelerator is KEY_F1  
end action bar
```

The above action bar template would produce the following figure after being referenced in the **action bar** attribute of a region definition.



10.4 Manipulating Action Bars and Items

10.4.1 Manipulating Action Bars

Action bar templates can be created at runtime or added at compile time. They cannot be modified once they have been created. However, you can use a different action bar template to create a new action bar for a region, delete a region's action bar altogether, or make dynamic changes to a region's action bar by manipulating its items.

10.4.1.1 Adding Action Bars at Runtime

add action bar adds a new template. The template itself is not an on-screen object; the action bar only appears when it is specified as the action bar of a region. The action bars in different regions can be created from the same template.

Model

```
add action bar AB_NAME is  
    [PULLDOWN_DEF_OR_REF]...  
    [BUTTON_DEF_OR_REF]...  
  
end action bar
```

Once a new action bar template has been added, it can be used to create any region's action bar.

10.4.1.2 Changing a Region's Action Bar

The action bar associated with a region can be changed dynamically. This is done by using a different action bar template to create the region's action bar.

Model

```
change { REGION_NAME | REG_CLASS_NAME } action bar to AB_NAME
```

This action can be used to change the action bar associated with an entire class of regions, if desired.

10.4.1.3 Deleting a Region's Action Bar

A region's action bar can also be removed:

Model

```
delete action bar from { REGION_NAME | REG_CLASS_NAME }
```

This action can be used to delete the action bar from an entire class of regions, if desired.

10.4.2 Manipulating Action Bar Items

Item definitions, like action bar definitions, cannot be changed at runtime. New item definitions can be added and then used in regions' action bars. The appearance of an item in a region's action bar can be changed at runtime. Items can be added and deleted from regions' action bars.

Whenever you perform an action on a pulldown, button, choice, or separator, you must name the region (graphical, textual, or image) on which to operate. Only the item in the specified region is affected. Optionally, you can specify the name of a class of regions, and the action will be performed on each of the members of the class.

None of these changes affect the item's definition. Only the item's appearance in the specified region or class of regions is changed. The action bar templates are not modified either.

10.4.2.1 Naming Items

Sometimes there may be more than one item with the same name. Each item with the same name must belong to a different action bar template. However, globally defined items always have unique names. In the example in 10.3.10, "An Example Action Bar Template" on page 10-17, the only globally defined item is `OpenFile`, which is defined outside the `MainMenu` action bar template. Any attempt to create another item named `OpenFile`, inside or outside an action bar template, will produce an error.

When referring to an item in an action statement or a response, you can usually just use the form **item** `ItemName`. However, sometimes there may be more than one item with the same name. Each such item must be referred to as **item** `ItemName` **from** `ActionBarName`. Whenever an item is defined in an action bar template, you can refer to it using **item** `ItemName` **from** `ActionBarName` although you are required to do so only when an item in another template has the same name.

Global items can be referred to only as **item** ItemName. In the example, the global item OpenFile cannot be referred to using **item** OpenFile **from** MainMenu although it appears in MainMenu.

10.4.2.2 Item classes

Items can be grouped together into **item classes**. Although item classes are similar to object classes, the two should not be confused. Object classes contain regular EASEL OS/2 EE objects. Object classes are used in place of objects or regions, to apply an action to more than one region at a time. Some of the actions and responses in this chapter can be used with object classes. **change action bar**, for example, when applied to an object class, changes the action bar of all of the regions in that class.

Item classes, however, contain a collection of pulldown, button, choice, and separator *definition* references. Item classes are used in place of items, to apply an action to more than one item at a time. Items from the same action bar or different action bars can be grouped into a class. Item definitions added at runtime must be added to classes in order to write responses for them.

Only the reference to the template can be added to classes; items from specific regions cannot.

Responses can be written for item classes as well as items, as the model in 10.5.1, “Responding to Items in a Region” on page 10-27 shows. Also, the functions that manipulate items (**check item** and **uncheck item**, **enable item** and **disable item**, and **change item text**) can be applied to entire item classes.

You may want to disable a large group of items all at once. For example, your action bar might contain a number of pulldown choices which the user cannot select until the OpenFile choice is selected. This is a good example of a use for item classes. If all of these items are placed in an item class together, then they can be enabled and disabled as a group.

10.4.2.3 Manipulating Item Classes

An item class is always referred to with the word **item** before the word **class**. The following model shows how to create a new item class.

Model

```
item class ITEM_CLASS_NAME [is  
    ITEM_NAME1 [from AB_NAME1] [ITEM_NAME2 [from AB_NAME2] ]... ]
```

Items can be dynamically added to item classes ...

Model

```
add item { ITEM_NAME [from AB_NAME] | ITEM_CLASS_NAME } to
  item class ITEM_CLASS_NAME
```

... or dynamically deleted from them.

Model

```
delete item { ITEM_NAME [from AB_NAME] | ITEM_CLASS_NAME }
  from item class ITEM_CLASS_NAME
```

Note: If an item was referenced as **item** ITEM_NAME **from** AB_NAME when the item was added to the item class, then it must be referenced the same way in the **delete item** statement. If **from** ActionBarName was not specified when the item was added to the item class, then it should not be specified in the **delete item** statement.

10.4.2.4 Checking and Unchecking Items

A choice may be created checked or unchecked. This status can be changed using the **check item** and **unchecked item** action statements. Again, the choice's definition is not changed, merely its status in a particular region or class of regions.

Model

```
{ check | unchecked } item
{ ITEM_NAME [from AB_NAME] | ITEM_CLASS_NAME }
  in { REGION_NAME | REG_CLASS_NAME }
```

A checked choice is displayed with a checkmark to its left.

10.4.2.5 Enabling and Disabling Items

An item may be created enabled or disabled. This status can be changed using the **enable item** and **disable item** action statements, which can be applied to pulldowns, choices or buttons in any region or class of regions.

Model

```
{ enable | disable } item
{ ITEM_NAME [from AB_NAME] | ITEM_CLASS_NAME }
  in { REGION_NAME | REG_CLASS_NAME }
```

When an action bar or pulldown item is disabled, it is grayed and cannot be selected. In this respect, it behaves differently than other EASEL OS/2 EE objects, which do not change appearance when disabled. This is because items are disabled by PM.

10.4.2.6 Changing the Text of an Item

The text of any item in a region's action bar can be changed dynamically to the value of any EASEL OS/2 EE string. The item's definition, however, will retain the original text; if another item is created using this item's definition, the text will appear as it was when the item was defined.

Model

```
change item { ITEM_NAME [from AB_NAME] | ITEM_CLASS_NAME }
  in { REGION_NAME | REG_CLASS_NAME }
  text to ITEM_TEXT
```

This action can also be applied to an item which has been created in the action bars of a class of regions.

10.4.2.7 Adding Item Templates at Runtime

Item templates can be dynamically created during runtime. They can then be added to item classes, or to action bars or pulldowns. These item definitions are global; they appear outside of any action bar template.

Model

```
add ITEM_DEF
```

This command adds an item template to the collection of item templates available to the EASEL OS/2 EE developer. No item will appear on the screen until it is added to a region with a visible action bar (see 10.4.2.8, "Adding Items to Action Bars" on page 10-24 and 10.4.2.9, "Adding Items to Pulldowns" on page 10-25).

Like any dynamically created EASEL OS/2 EE object, an item must be added to a predefined *item class* in order to trigger a response.

10.4.2.8 Adding Items to Action Bars

Pulldowns and buttons can be added dynamically to any region's action bar. The pulldown or button must be defined globally, either at compile time or by adding it at runtime. Once the item definition exists, it can be used to create a new item in the action bar of a region or class of regions.

Model

```
add item ITEM_NAME1 to action bar
  in { REGION_NAME | REG_CLASS_NAME }
  [ { before | after } item ITEM_NAME2 [from AB_NAME] ]
```

All the pulldowns in an action bar should appear to the left, and the buttons should appear to the right.

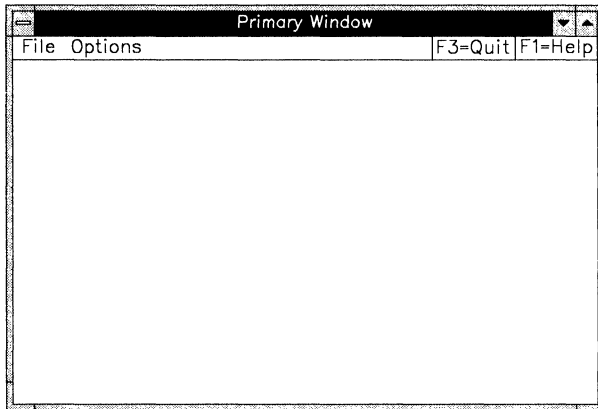
Note: If you have buttons in an action bar, add any pulldowns before the first button item.

The **before** and **after** keywords are used to position the new pulldown or button as desired. **before** positions the new item to the left of the existing item, while **after** positions the new item to the right of the existing item.

The following example dynamically adds a quit button to the MainMenu action bar we defined earlier.

```
add button QuitButton text "F3 = Quit"
  accelerator is KEY_F3
add item QuitButton to action bar in Region
  before item HelpButton
```

Note that QuitButton is added only to the action bar in a single region or class of regions. The action bar template MainMenu, which was used to create the action bar in Region, does not change.



10.4.2.9 Adding Items to Pulldowns

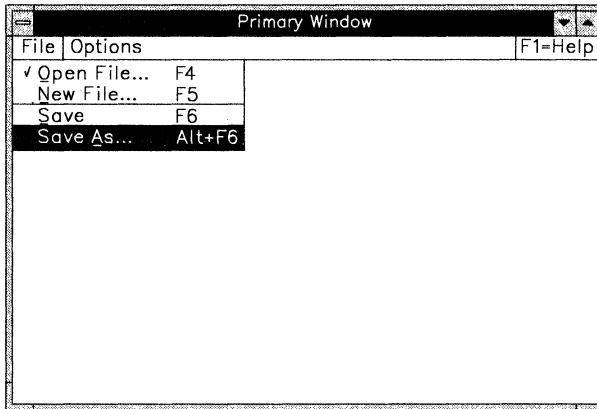
Choices and separators can be added dynamically to a pulldown in any region or class of regions. The item must be defined globally, either at compile time or by adding it at runtime. Once the item definition exists, it can be used to create a new item in a pulldown in the desired region or class of regions.

Model

```
add item ITEM_NAME1 to
  pulldown PD_NAME [from AB_NAME]
  in { REGION_NAME | REG_CLASS_NAME }
  [ { before | after } item ITEM_NAME2 [from AB_NAME] ]
```

ITEM_NAME1 and ITEM_NAME2 refer to choices and separators that are added to the specified pulldown **before** (above) or **after** (below) the specified items. In order to add a separator to a pulldown dynamically, it must be named. The following example dynamically adds a separator at runtime, then adds it to the FileMenu pulldown to separate the Save functions.

```
    add separator Sep
    add item Sep to pulldown FileMenu in Region before
item SaveFile
```



10.4.2.10 Deleting Items from Regions

Just as items can be added to regions dynamically, they can also be deleted.

Model

```
delete item { ITEM_NAME [from AB_NAME] | ITEM_CLASS_NAME }  
  in { REGION_NAME | CLASS_NAME }
```

delete item can also be applied to a class of regions in which each region contains the item to be deleted.

If a separator is to be deleted using this command, it must have a name.

10.5 Responding to Action Bar Items

Action bar input is handled through the normal EASEL OS/2 EE response mechanism. A response may be written for any button in an action bar or for any choice in a pulldown. You cannot write a response to an action bar or pulldown. When an enabled item is selected—by clicking on it with the mouse, by using its accelerator key, mnemonic, or pressing ENTER while the item is active—its response is taken.

10.5.1 Responding to Items in a Region

When writing a response to a button or choice, you must provide the **item** keyword.

Model

```
[interrupt] response to item
{ ITEM_NAME [from AB_NAME] | ITEM_CLASS_NAME }
  [in { REGION_NAME | CLASS_NAME } ]

  [ACTION_STATEMENT]...
```

Item responses can appear in blocks just like any other response. As is the case with all EASEL OS/2 EE objects, selecting an item which has no response has no effect.

By default, an item response refers to the item definition. This means that the item response will be taken whenever an item created from this definition is selected. The response will apply to items defined or referenced in action bar templates as well as those added dynamically to pulldowns or action bars.

If the optional **in** keyword is specified, then the response is only taken when the region containing the selected item is either `REGION_NAME` or is in the region class `CLASS_NAME`.

If you want to write a generic response to an item, and replace it with a specific response only in a region or regions, make sure to place the generic response after the specific response in the EASEL OS/2 EE program.

10.6 Built-in Functions for Action Bar Items

10.6.1 Built-in Functions in Inquiries

Two built-in functions are provided to inquire about the status of an item.

You can use the **is enabled** built-in function to inquire if an item is enabled (selectable).

Model

```
( item ITEM_NAME [from AB_NAME] in REGION_NAME is enabled )
```

You can use the **is checked** built-in function to inquire if an choice is checked.

Model

```
( item ITEM_NAME [from AB_NAME] in REGION_NAME is checked )
```

The entire statement must be enclosed in parentheses.

The value returned is the boolean (**true** or **false**) associated with the enabled or check state of the specified object. If the object is enabled or checked, the value **true** is returned; if it is disabled or unchecked, the value **false** is returned.

You can use the **text of item** built-in function to get the text associated with an item.

Model

```
text of item ITEM_NAME [from AB_NAME] in REGION_NAME
```

An EASEL OS/2 EE string containing the current text of the item is returned.

10.6.2 Built-in Functions in Responses

During an item response, certain built-in variables are available. The built-in variables **object** and **ancestry** contain, respectively, the name of the item selected and the name of the region affected.

Chapter 11. Program Flow and Performance

11.1 Controlling Program Flow

The following statements allow you to control the flow of an EASEL OS/2 EE interface, which may consist of one or more EASEL OS/2 EE programs.

action	Invokes an action routine.
action is	Defines an action routine.
call	Invokes a subroutine.
change to program	Transfers control from the current program to another program.
exit	Terminates the current program.
for each member loop	Specifies a set of action statements to be performed for each member of a class.
for each selected line loop	Specifies a set of action statements to be performed for each selected line in a list box.
for loop	Specifies a set of action statements to be performed repeatedly, until an integer variable reaches a specified value.
function	Declares an EASEL OS/2 EE external function.
if	Tests whether a specified condition is true or false, and allows you to perform actions based upon the result of the test.
include	Includes another source code file in the current program, at compile time.
invoke	Allows you to execute another application while EASEL OS/2 EE is running.
leave loop	Allows you to branch out of a loop.
save program as	Creates a new program by saving the currently executing program with the current state of its objects and the current values of its local variables.
subroutine	Declares an internal EASEL OS/2 EE subroutine or an external subroutine.
subroutine is	Defines an EASEL OS/2 EE subroutine.
switch	Performs pattern matching on a specified string value.

while loop

Allows you to specify actions to be performed repeatedly, as long as a specified condition is true.

11.2 Functions, Subroutines, and Action Routines

11.2.1 Declaring External Subroutines and External Functions

EASEL OS/2 EE supports the use of two different types of subroutines within a program:

- External subroutines
- Internal EASEL OS/2 EE subroutines

External subroutines and external functions are written in a language other than EASEL OS/2 EE and are compiled into an OS/2 dynamic link library. Before either an external subroutine or function can be used in an action statement, it must be declared so that EASEL OS/2 EE knows what argument types it needs and what library it is in. For example:

```
subroutine MIN(float: ArrayIn, float: Minimum)
    library "mathlib"
```

```
function WeekDay(integer:Year, integer:Month,
                  integer:Day)
    returns string
    library "dateLib"
```

Libraries provided with EASEL OS/2 EE also have include files that contain declarations for all the subroutines and functions in the library. See 25.2, "Writing Library Functions and Subroutines" on page 25-3 for information on writing library routines to be used with EASEL OS/2 EE; see Chapter 13, "Library Subroutines and Functions" for more information on Libraries.

11.2.2 Action Routines and Internal EASEL OS/2 EE Subroutines

EASEL OS/2 EE action routines and internal EASEL OS/2 EE subroutines are used to define a set of action statements that will be performed in many places throughout the EASEL OS/2 EE program. You can use these constructs to avoid having to specify the same set of action statements many times within a program. You cannot define another action routine or subroutine within a definition. However, you can invoke action routines or subroutines from within another action routine or subroutine, provided they have been defined previously or provided the subroutine has been declared previously. With EASEL OS/2 EE subroutines, you can also use local variables and parameters.

Action routines and EASEL OS/2 EE subroutines are not global; they must be defined in every program that references them. To save time and space, you can put them in a separate file and include the file in the program. See

11.3, “General Program Control” on page 11-8 for information on the **include** statement.

11.2.2.1 Defining an Action Routine

The **action is** statement allows you to group a set of action statements into an action routine with its own identifier. The following is the model for an action routine definition.

Model

```
action ROUTINE_NAME is  
  
    [ACTION_STATEMENT]...
```

For example:

```
action NextProgram is  
    make Everything invisible      # Erase screen.  
    make PleaseWait visible       # Inform user.  
    change to program Program     # Program variable contains  
                                  # actual name.  
    :
```

Note the following special characteristics of action routines.

- All EASEL OS/2 EE variables, expressions, and built-in EASEL OS/2 EE functions are evaluated when an action routine is executed, not when it is defined. Thus, an expression in the routine that has a certain value the first time the routine is referenced might have a different value the next time the routine is referenced.
- Any action statement can be specified in an action routine definition, even a reference to another action routine.
- Any number of action statements can be specified in an action routine.

11.2.2.2 Declaring an Internal EASEL OS/2 EE Subroutine

An internal EASEL OS/2 EE subroutine can be declared and then referenced before it is defined. The following is the model for an internal EASEL OS/2 EE subroutine declaration.

Model

```
subroutine SUB_NAME( [TYPE1:ARG1_NAME[,  
                     TYPE2:ARG2_NAME]... ] )
```

Unlike an EASEL OS/2 EE subroutine definition, a declaration does not have an **is** statement, local variables, or action statements. For example, the following is a declaration:

```
subroutine DrawGraph(string:RegionName,  
                    string:GraphType)
```

The following is the actual definition, which can appear anywhere in the program after the declaration:

```
subroutine DrawGraph(string:RegionName,  
                    string:GraphType) is  
    integer A  
  
    copy RegionName to GRegion      # Region used by  
                                    # BGraph module  
  
    if (GraphType = "Bar") then  
        action GBars  
    end if  
    :
```

When EASEL OS/2 EE sees the actual definition, and a declaration has already been made, it checks to make sure the type and number of the arguments are the same. If they are not, it generates a compile-time error. If there is an internal EASEL OS/2 EE declaration but no definition, a runtime error is generated when the subroutine is called.

11.2.2.3 Defining an EASEL OS/2 EE Subroutine

The **subroutine is** statement allows you to group a set of action statements into a subroutine with its own identifier, argument list, and local variables. The following is the model for an EASEL OS/2 EE subroutine.

Model

```
subroutine SUB_NAME( [TYPE1:ARG1_NAME[,  
                    TYPE2:ARG2_NAME]... ] ) is  
    [LOCAL_VAR]...  
  
    ACTION_STATEMENT  
    [ACTION_STATEMENT]...
```

For example:

```
subroutine ReadFile(string:InputFile,  
                   string:DisplayRegionName) is  
    read file InputFile into DisplayRegionName
```

EASEL OS/2 EE subroutines perform in much the same way as action routines, except for the following distinctions:

- EASEL OS/2 EE subroutines have an argument list as part of their syntax. This allows you to pass parameters to the subroutine in the form of variables, which must be previously declared in your program before being passed as arguments. The arguments passed must match those declared for the subroutine in type and number. Note that you cannot pass constants or literals as arguments. The values of the argument list variables can be changed by the subroutine, and the new values will also be reflected by the value of the variable used as the argument, even outside of the subroutine.
- EASEL OS/2 EE subroutines support the use of local variables. Thus, you can declare variables within a subroutine to be used by that subroutine only. You cannot define constants within a subroutine. Storage space for local variables is allocated only when the subroutine is entered, and is freed when the subroutine has completed execution.

11.3 General Program Control

11.3.1 Including Another EASEL OS/2 EE File in the Current Program

The **include** statement inserts a specified ASCII text file, which contains EASEL OS/2 EE source code, into the current EASEL OS/2 EE program. The file is inserted at the time that the program is compiled. You can specify any number of **include** statements in an EASEL OS/2 EE program.

Model

```
include FILE_NAME
```

This feature is especially useful when you want to use the same sequence of statements in more than one program. For example, you might define a standard set of keys that are displayed in all of your programs. Instead of specifying identical statements in each of the programs, you can put the statements into a separate file—for example, one named:

```
keys.inc
```

Then, at the appropriate line in each of your programs, you can simply specify:

```
include "keys.inc"
```

EASEL OS/2 EE inserts all of the statements contained in the included file at the location at which the **include** statement is specified. After processing the last line of the inserted file, EASEL OS/2 EE continues to process the original file at the line following the **include** statement line.

The string literal that you specify in the **include** statement must be an OS/2 file name. Although the examples shown in this section use the extension .INC (for INCLUDE), you can specify any other extension. If you have specified a value for INCTYPE in the CONFIG.ESL file, it will be used when no extension is specified on the **include** command line.

11.3.1.1 Nested include files

Included files can be nested; that is, each file can contain **include** statements. For example, suppose that a program contains the statement:

```
include "graphics.inc"
```


The GRAPHICS.INC file might contain the statement:

```
include "boxes.inc"
```

The BOXES.INC file might contain the statement:

```
include "points.inc"
```

You can nest **include** statements to 20 levels. Make sure that no file includes itself or any other file that has already been included. In the example above, the POINTS.INC file must not specify **include** statements for the GRAPHICS.INC or BOXES.INC files. If it does, each of the two files will cause the other file to become part of it each time EASEL OS/2 EE encounters the **include** statement, and inclusion will never end.

EASEL OS/2 EE generates error messages if you specify an invalid file name or too many levels of nesting, or if the specified file cannot be accessed or opened.

11.3.2 Exiting the Program

There are two ways to leave an EASEL OS/2 EE program. You can specify an **exit** statement to return control to the operating system, or a **change to program** statement to transfer control to another EASEL OS/2 EE program within the complete EASEL OS/2 EE interface.

Model

```
exit [EXIT_CODE]
```

The **exit** action statement ends the current program. The **exit** action statement, like other action statements, can be specified in a response definition or in an action subroutine definition.

The **exit** statement should be specified as the final statement in the set of action statements in the response definition for terminating the program. Statements which follow the **exit** statement will be ignored. For example:

```
response to ExitKey
    send ShutdownMessage to Audit
    exit
```

In this example, if **exit** were specified prior to the **send** statement, EASEL OS/2 EE would terminate prior to executing the **send** statement, and the Audit program would not receive its logoff code.

You can optionally specify an integer value as the EXIT_CODE in the **exit** statement; this value is returned to the operating system when EASEL OS/2 EE terminates. For example:

```
response to ExitKey
    send ShutdownMessage to Audit
    exit 32
```

This value provides information about the EASEL OS/2 EE session and might be used by the operating system to condition the execution of other programs.

If you wish to transfer control to another EASEL OS/2 EE program, rather than exit from EASEL OS/2 EE entirely, you can use the **change to program** statement. If you wish to save the currently executing program as a new program with the current state of its objects and local variables, you can use the **save program as** statement prior to the **exit** or **change to program** statement.

11.3.3 Invoke

The **invoke** action statement allows you to execute another application while EASEL OS/2 EE is running. You specify the full OS/2 filename of the application to execute, including the extension (which is always ".exe") and any options the program requires. EASEL OS/2 EE continues to run while the invoked program is running.

Model

```
invoke PROGRAM_NAME [COMMAND_LINE_OPTIONS]
```

You should use **invoke** only while EASEL OS/2 EE is running within a window rather than in fullscreen mode because switching input focus between the invoked program's window and EASEL OS/2 EE's fullscreen is very cumbersome.

Some applications, when invoked, might display "ESLRUN.EXE" in their title bar instead of their own name.

If starting a non-PM application, the application will run in its own OS/2 fullscreen session in text mode.

11.3.3.1 Example

```
invoke "textedit.exe" "c:\\document\\memo5.doc"
```

11.3.4 Transferring to Another Program

If your interface consists of multiple EASEL OS/2 EE programs you must specify a **change to program** action statement to tell EASEL OS/2 EE when to transfer control from one program to another. (If you wish to exit from EASEL OS/2 EE completely, rather than transfer control to a new program, use the **exit** statement.)

Model

change to program PROGRAM_NAME

When EASEL OS/2 EE executes a **change to program** statement, it causes the current program to terminate. All objects, variables, constants, classes, patterns, and responses belonging to the program that is being left cease to exist. Note, however, that global variables still exist, started applications continue to run, and all input devices maintain their current state. The new program begins execution with its **response to start** statement.

The **change to program** statement, like other action statements, can be specified in a response definition or in an action subroutine definition. Any statements specified after the **change to program** statement will only be executed if the **change to program** statement could not be performed, for instance if the new program could not be found. You might specify the following response definition:

```
response to Database_Key
  send ExitCode to CurrentApp
  change to program "Database"
  copy "Could not find requested program" to InfoString
  call DisplayError(InfoString)
```

Here, Database is a string literal identifying the name of an EASEL OS/2 EE program that interfaces a database retrieval application; EASEL OS/2 EE transfers control to that program. Note that the .EBI extension is not required. The copy and subroutine call to DisplayError will not be performed unless the change to program statement was unsuccessful.

The value for PROGRAM_NAME must be a string value and may contain a pathname. When specifying a path, include an extra backslash each time one is specified; for example:

```
change to program "c:\\easel-ee\\myfile"
```

This is because a backslash *by itself* within an EASEL OS/2 EE string literal indicates an escape sequence, telling EASEL OS/2 EE to treat the next character as a special control character.

11.3.5 Saving a Program in Progress

When a program is exited with an **exit** or a **change to program** action statement, any changes that took place during execution are abandoned; when that program is run again, it is run exactly as it was written. But by specifying the **save program as** action statement before any **exit** or **change to program** statement is encountered, you can save the program, with the changes made during execution, to a new EASEL OS/2 EE binary program file. (This statement does *not* terminate the currently executing program.) Like other action statements, this statement can be specified in a response definition or an action subroutine definition.

Model

```
save program as PROGRAM_NAME
```

When an EASEL OS/2 EE program is executed, changes might be made to object attributes and/or to the local variable values; new objects might be created, or the composition of classes might change. For example, when Square is selected, the key Bubble, which was red, is changed to the color green:

```
red key Bubble ...

response to Square
  make Bubble green
  exit
```

but Bubble will be red, not green, when the program is run a second time. In contrast, the **save program as** statement saves the program in its current state. All objects are saved with their current appearance, attributes, and text and graphics cursor positions; local and global variables are saved with their current values. Thus, if a **save program as** statement is executed prior to the **exit** statement, such as:

```
red key Bubble ...

response to Square
  make Bubble green
  save program as "revised.ebi"
  exit
```

If Square is selected, Bubble will be green, not red, when the program REVISED.EBI is run. Any other changes that were made during execution of the program will also be saved in the REVISED.EBI file.

When you save a program, the program currently executing still exists as it was written, and continues to run until an **exit** or **change to program** statement is executed. The new program, as amended during execution, has

been saved in a new file, identified by the specified value for `PROGRAM_NAME`. If you are going to transfer control to this binary program file, be sure that the name includes the **.EBI** extension. But be careful: if you specify a **save program as** statement with the same name as any other existing EASEL OS/2 EE binary file, including the one that is currently being executed, the saved program will overwrite the original program.

The saved program does not have to be in the current directory; you can specify a full path name for it. When specifying a path, include an *extra* backslash each time one is specified; for example:

```
save program as "c:\\ease1-ee\\second.ebi"
```

since a single backslash is the EASEL OS/2 EE escape character in a string literal.

11.4 Global and Local Entities

A *global* entity is one that is shared among multiple programs of an EASEL OS/2 EE interface. EASEL OS/2 EE provides the following global entities, which reside in global memory:

- Variables (including variable arrays), when defined as **global**
- Applications (remote and local)
- Pointing devices

All other entities in an EASEL OS/2 EE program are *local* to that program. For example, a value assigned to a local variable in a program will be retained as the value of that variable only as long as that program is executing. When EASEL OS/2 EE transfers to another program, the variable will no longer exist.

11.4.1 Variables

You can define and reference *global* variables in multiple programs of an EASEL OS/2 EE interface. The value of the variable in the first program is retained when control is transferred to another EASEL OS/2 EE program.

Every program that uses global variables must define them as such, with the same names and the same types.

A value assigned to a *local* variable in a program will be retained only as long as that program is executing; it ceases to exist when EASEL OS/2 EE leaves the program in which the variable is defined.

11.4.2 Applications

All applications are global. In one program, you might define an application and start it running. In the other programs to which control is transferred, you define the application but you do not restart it. The application continues to run without interruption as programs are changed. 14.1.2, "Defining, Starting, and Stopping Applications" on page 14-3 discusses the definition and use of applications.

11.4.3 Devices

All pointing devices are global. If you open a device (e.g, a mouse) in one program, the device will remain open when control is transferred to another program. You do not need to reopen it.

11.4.4 Memory

Global memory is specified with the GLBSIZE configuration option; local (program) memory is specified with the PRGSIZE configuration option.

11.4.5 Global Entities in Changed Programs

Make sure that any program which will be run via the **change to program** statement does not initialize any global variables, since these initializations will be ignored. Even when control is transferred to a program which has been previously saved (with the **save program as** statement, discussed below), the **change to program** statement transfers the value of its current global variables to the new program. The following code illustrates this situation:

```
global integer variable MaxCount is 10
```

```
response to ...
```

```
    copy 20 to MaxCount
```

```
    save program as Revised      # MaxCount in Revised is 20
```

```
    copy 30 to MaxCount
```

```
    change to program Revised    # MaxCount in Revised is 30
```

When global arrays are loaded from global memory into local memory for a new program, the new number of dimensions in the array is *not* checked against the old number. Therefore, if you execute a **change to program** for programs that contain global arrays, try to ensure that the array dimensions in the all programs are the same. If the new program array dimensions are larger than as defined in the original program, the last elements will contain default values for the constant or variable types; if the new program's array dimensions are smaller than as defined in the original program, the last elements will get dropped.

11.4.6 Global and Local Values in Saved Programs

When a program is saved with a **save program as** statement, the values of global *and* local variables are saved with it. For example:

```
integer Min is 10
```

```
global integer NewMax is 20
```

```
response to ChangeSize
```

```
    copy 15 to NewMax
```

```
    copy 13 to Min
```

```
response to StopSign
```

```
    save program as "revised.ebi"
```

After ChangeSize and StopSign have been selected, the value of the local variable Min will be 13, and the value of the global variable NewMax will be

15. These values will be used when the binary file created by the **save program as** statement is run, unless control is transferred to it from another program (with the **change to program** statement) which has global variables. If that program contains the global variable **NewMax**, the saved value will be overwritten.

11.5 Controlling Flow Within a Program

The following action statements are provided for controlling program flow:

action	Specifies a set of action statements should be executed.
if/then/else	A conditional action statement; specifies action statements to be performed based on whether a specified condition is true or false.
call	Specifies either an external or EASEL OS/2 EE internal subroutine should be run.
for loop while loop	Specifies action statements to be performed repeatedly, as long as a specified condition is true.
for each member loop	Specifies action statements to be performed for each member of a class.
for each select line loop	Specifies action statements to be performed for each selected line in a list box.
leave loop	Explicitly exits from a loop.
switch	Performs pattern matching on a specified string value.

11.5.1 Referencing an Action Routine

An action routine is invoked with the **action** action statement. The specified routine must previously have been defined in the EASEL OS/2 EE program. When the routine is invoked, EASEL OS/2 EE performs the action statements specified in the subroutine, just as if the actions had been specified directly in the response definition. The routine name that you specify in an **action** statement can be either an EASEL OS/2 EE name or a string value. If a string value is specified, EASEL OS/2 EE converts this value using the string/name conversion rules summarized in 4.9.4, "String/Name Conversions" on page 4-55. In the following example, note that the name of the routine referenced by the **action** statement is extracted from the value returned by the **parameter** function. This function returns different values depending upon which key the user selects; for example:

```
response to KeyClass
  extract from parameter # The parameter contains the
    take word ActionName # name of an action
  action ActionName      # routine.
```

You can invoke an action routine inside the definition of another action routine. However, you must ensure that an action routine does not invoke itself—that is, the **action** statement inside the routine definition must not specify the name of the routine in which it is defined.

11.5.2 Specifying a Conditional Action Statement

EASEL OS/2 EE allows you to specify a set of action statements to be performed if a given condition is true, and other alternative action statements to be performed if the condition is false. You specify both the condition and the actions in a conditional action statement. A conditional action statement must begin with the **if** statement; it must contain a **then** clause; and it must end with the **end if** keywords. It can optionally contain an **else** clause. This is shown in the following syntax model.

Model

```
if ( BOOLEAN_VALUE ) then  
    [ACTION_STATEMENT] ...  
  
[else  
    [ACTION_STATEMENT] ... ]  
  
end if
```

Like other action statements, a conditional action statement can be specified within a response definition or an action routine definition. It can be specified within a **loop** statement, or even within another conditional action statement.

EASEL OS/2 EE first evaluates the boolean value specified in the **if** statement. The result of every boolean is either **true** or **false**. Remember that any boolean expression must be enclosed in parentheses.

If the boolean value is **true**, EASEL OS/2 EE performs the action statements specified in the **then** clause; the **else** clause (if specified) is ignored. If the boolean value is **false**, EASEL OS/2 EE performs the action statements specified in the **else** clause (if specified); the **then** clause is ignored.

After executing the **then** clause or the **else** clause, EASEL OS/2 EE continues normal program execution at the action statement (if any) immediately following the **end if** keywords. Consider the following example:

```
response to Password
  if (Code = "BN") then
    copy 1 to Ncode
    make Menu2 invisible
    make Menu1 visible
  else
    copy 2 to Ncode
    make Menu1 invisible
    make Menu2 visible
  end if
  send Ncode to Dbase
```

Here, EASEL OS/2 EE first evaluates the boolean expression:

```
(Code = "BN")
```

If the variable Code contains the value "BN", the expression is **true**, and EASEL OS/2 EE executes the **then** clause and then ignores the **else** clause.

```
  copy 1 to Ncode
  make Menu2 invisible
  make Menu1 visible
```

If the Code variable does not contain the value "BN", the expression is **false**, and EASEL OS/2 EE ignores the **then** clause and executes the **else** clause:

```
else
  copy 2 to Ncode
  make Menu1 invisible
  make Menu2 visible
```

In either case, EASEL OS/2 EE continues execution at the statement following the **end if** keywords. Thus, in the example, EASEL OS/2 EE performs the following action statement whether or not Code contains "BN":

```
send Ncode to Dbase
```

This statement causes the value of the Ncode variable to be sent to the application program named Dbase.

As shown in the model, the **else** clause is optional. For example, you can specify:

```
if ( (A > B) and (Count < MaxCount) ) then
  append Count to Log
  send Count to LogProg
end if
```

In this case, if the boolean value is **false**, EASEL OS/2 EE ignores the **then** clause and continues execution at the statement following the **end if** keywords.

11.5.2.1 Nesting Conditional Action Statements

You can nest conditional action statements to any number of levels; that is, you can specify conditions within other conditions. Note that you must specify the **end if** keywords corresponding to each **if** test. For example:

```
if (TestSuccessFlag) then
  if (A > B) then
    copy A to K
    send A_Msg to Recorder
  else
    send B_Msg to Recorder
  end if
end if
copy A to Log
```

EASEL OS/2 EE initially evaluates the boolean in the first **if** statement. If the value of the TestSuccessFlag variable is **true**, EASEL OS/2 EE executes the **then** clause. The first action statement in the **then** clause is another **if** statement:

```
if (A > B) then
```

EASEL OS/2 EE evaluates this boolean. If A is greater than B, EASEL OS/2 EE executes the **then** clause:

```
copy A to K
send A_Msg to Recorder
```

EASEL OS/2 EE ignores the **else** clause and continues execution at the statement following the second **end if**:

```
copy A to Log
```

If A is not greater than B, EASEL OS/2 EE executes the **else** clause:

```
send B_Msg to Recorder
```

EASEL OS/2 EE ignores the **then** clause and continues execution at the statement following the second **end if**:

```
copy A to Log
```

If the value of the TestSuccessFlag variable is **false**, the result of the first boolean is **false**, and EASEL OS/2 EE ignores the entire outer **then** clause. Because there is no **else** clause corresponding to this condition, EASEL OS/2 EE continues execution at the statement following the second **end if** (copy A to Log).

11.5.3 Specifying a Call Action Statement

In order to invoke an external or EASEL OS/2 EE subroutine you use the **call** action statement. In the **call** statement you specify the name of the variables you want to pass as arguments to the subroutine. For example:

```
call MIN(MyNumbers, TheMin)
```

where MyNumbers is a floating point array name which contains the numbers to be compared. The minimum value is returned in the floating point variable TheMin. The argument variables used in the **call** statement must be the same number and type as those specified in the subroutine declaration or definition.

Model

```
call SUBROUTINE_NAME( [ARG1[, ARG2]... ] )
```

External functions are invoked by referencing the function name, followed by its arguments enclosed in parentheses, wherever a value of the same type as the return type can be used.

11.5.4 Specifying a for or while loop

A **loop** allows you to specify a set of action statements to be performed repeatedly, as long as a specified condition is true. You specify both the condition and the actions in the statement. The **loop** statement must begin with the **for** or **while** clause—which must be followed by the **loop** keyword—and it must end with the **end loop** keywords. This is shown in the following syntax models.

Model (for loop)

```
for VAR_NAME = START_VAL to END_VAL [by INCR_VAL] loop [LOOP_NAME]
    [ACTION_STATEMENT]...
    [leave loop [LOOP_NAME] ]
end loop [LOOP_NAME]
```

Model (while loop)

```
while ( BOOLEAN_VALUE ) loop [LOOP_NAME]
    [ACTION_STATEMENT]...
    [leave loop [LOOP_NAME] ]
end loop [LOOP_NAME]
```

Like other action statements, the **loop** statement can be specified within a response definition or an action routine definition.

For **while** loops, EASEL OS/2 EE first evaluates the **BOOLEAN_VALUE** specified in the **while** clause. The result of every boolean is either **true** or **false**. Remember that any boolean expression must be enclosed in parentheses.

If the boolean value is **true**, EASEL OS/2 EE performs the action statements specified in the loop. When EASEL OS/2 EE reaches the **end loop** keywords, it evaluates the boolean once again. If the result is **true**, EASEL OS/2 EE returns to the top of the loop and performs the action statements again. This cycle continues for as long as the boolean value is **true**.

If the boolean value is **false**—during either the initial evaluation or a later evaluation—EASEL OS/2 EE skips the loop and continues execution at the action statement (if any) immediately following the **end loop** statement. If the boolean value is initially **false**, the loop will not be executed at all.

For **for** loops, EASEL OS/2 EE first sets **VAR_NAME** to the initial value **START_VAL**. It then compares the value of **VAR_NAME** with **STOP_VAL**. If **VAR_NAME** is not equal to **STOP_VAL** then the loop is executed. When EASEL OS/2 EE reaches the **end loop** statement, it increments **VAR_NAME** either by 1, which is the default increment value, or by **INCR_VAL** if you have specified one. If the new value of **VAR_NAME** is not equal to **STOP_VAL**, EASEL OS/2 EE executes the action statements again. The loop continues until **VAR_NAME** equals **STOP_VAL**.

A negative value can be used for INCR_VAL to cause VAR_NAME to go from a larger to a smaller value.

You can exit explicitly from a loop by specifying a **leave loop** statement, discussed below.

If INCR_VALUE is zero, the loop is not executed. If the current value is less than STOP_VAL and INCR_VALUE is negative, the loop is terminated. Similarly, if the current value is greater than STOP_VAL and INCR_VALUE is positive, the loop is terminated.

You can optionally name a loop. The identifier that you use to name a loop must be unique within the EASEL OS/2 EE program; there must be nothing else in the program with that name. If you specify a name in the **loop** statement, you must specify the same name in the **end loop** statement. Loop names are particularly useful if you intend to nest loops, as discussed below. They help you in checking that all of your **loop** and **end loop** specifications match correctly.

11.5.5 for each member loop

You can specify a loop which loops through all of the members of a class. The action statements specified in the loop will be executed as many times as there are objects in a class.

Model

```
for each member of CLASS_NAME loop [LOOP_NAME]

    [ACTION_STATEMENT]...
    [leave loop [LOOP_NAME] ]

end loop [LOOP_NAME]
```

The built-in function **member** is used to access the name of the current class element.

11.5.5.1 Accessing the Current Class Element

Use the **member** built-in function to access the current class element in a **for each** loop. EASEL OS/2 EE returns a string value representing the name (identifier) of the member.

member

For example:

```
copy false to InClass
for each member of CLASS loop
    if (member = OBJ1) then      # check to see if OBJ1 is in
        copy true to InClass    # this class
    leave loop
end if
end loop
```

The loop will progress through the objects in the class in the reverse of the order they were added; the last added will be the first member.

11.5.6 for each selected line loop

You can specify a loop which loops through all the selected lines in a list box. The action statements specified in the loop will be executed as many times as there are selected lines in a list box.

See 9.3.7.1, "Actions for List Boxes" on page 9-12 for details on **for each selected line**. See 9.3.7.2, "Built-in Functions for List Boxes" on page 9-14 for details on **selected line**, which is used for accessing the current selected line.

11.5.7 Exiting from a Loop

If you specify the **leave loop** statement as one of the action statements in a **loop** statement, EASEL OS/2 EE will exit from the loop before the condition specified in the **while** or **for** statement has failed, and will continue execution at the statement following the **end loop** keywords. For example, suppose you specify:

```
response to ...
while (Int <= 5) loop
    action Action_Sub
    if (Action_flag != "OK") then
        leave loop
    end if
    copy (Int + 1) to Int
end loop
copy Action_flag to Flag_log
```

EASEL OS/2 EE repeatedly performs the action statements specified in the loop until the first test condition fails. However, if the value of Action_flag (a

variable set by the action routine, Action_Sub) is not "OK" at any point, EASEL OS/2 EE immediately leaves the loop. Execution continues at the statement following the end loop:

```
copy Action_flag to Flag_log
```

If you specify a **leave loop** statement, you must specify it within the definition of the loop itself. You cannot specify it in an action routine that is called from inside the loop. In the example shown above, the statement:

```
if (Action_flag != "OK") then
    leave loop
end if
```

could not be specified within the Action_Sub subroutine.

You can optionally specify a name in the **leave loop** statement. This name might identify the loop in which the **leave loop** is specified, or it might identify an outer loop. If you specify an outer loop, EASEL OS/2 EE leaves the specified outer loop and all inner loops. If you do not specify a name in the **leave loop** statement, EASEL OS/2 EE exits from the innermost loop that contains the **leave loop** statement, even if the innermost loop has a name.

11.5.8 Nested Loops

You can specify loops within other loops, to any number of levels. The following example illustrates the use of loop names in nested loops, and shows how you can specify a conditional action statement within a loop.

```
response to ...
copy 1 to Int
while (Int <= 5) loop Count      # Outer loop
    for Action_num = 1 to 10 loop Action #Inner loop
        action Action_sub
        if (Action_flag = "OK") then      # Conditional
            send Action_flag to Recorder # action
            action Logit                  # statement
        else
            action Fixit
        end if
    end loop Action                # End of inner loop
    copy (Int + 1) to Int
end loop Count                    # End of outer loop
```

EASEL OS/2 EE performs the entire Count loop repeatedly, until the value of Int exceeds 5 and thus the first boolean expression is **false**. Within the Count loop, EASEL OS/2 EE performs a second loop, named Action. The statements in this loop are performed repeatedly, until the the value of Action_num exceeds 10. The Action loop is performed 10 times in each iteration of the Count loop, for a total of 50 times.

Within the Action loop, each time through the loop, EASEL OS/2 EE performs the conditional action statement that begins with the **if** statement:

```
if (Action_flag = "OK") then
```

Depending on the value of this expression, EASEL OS/2 EE performs the action statements in either the **then** or the **else** clause.

After completing each cycle of the Action loop, EASEL OS/2 EE returns to the top of the Action loop, increments Action_num and checks to see if the value of Action_num is not greater than 10, EASEL OS/2 EE performs the loop again. This cycle continues until the value of Action_num is greater than 10. At this point, EASEL OS/2 EE transfers to the statement following the Action loop:

```
copy (Int + 1) to Int
```

After executing these statements, EASEL OS/2 EE returns to the top of the Count loop and evaluates the boolean expression (Int <= 5) again. If the value of Int does not exceed the value of 5, EASEL OS/2 EE performs the entire loop again.

11.5.9 Blocks and Loops

You can specify blocks in loops, and vice versa. If you specify a block within a loop, execution of a **leave loop** statement causes all blocks contained in the loop to become inactive.

11.5.10 Switch Statement

The **switch** statement is used to perform pattern matching on a specified string value. It is particularly useful in situations where you have many possible values for a string and where nested **if/then/else** statements would be cumbersome.

Model

```
switch STRING_VALUE is
  case { char | line } MATCH_CLAUSE is

    [ACTION_STATEMENT]...

  [case { char | line } MATCH_CLAUSE is

    [ACTION_STATEMENT]... ]...

  default is

    [ACTION_STATEMENT]...

end switch
```

STRING_VALUE is an expression that is converted to a string value.

MATCH_CLAUSE is a clause describing the input to be matched for invoking the case, following the syntax model:

Model

```
[col NUM1 [thru NUM2] ] { STRING | word | number | blank }
```

NUM1 and NUM2 are compile-time integer values in the range 1-254, representing column positions in the input stream.

When a **switch** statement is executed the STRING_VALUE is compared with the MATCH_CLAUSE is each statement. The MATCH_CLAUSE is in the format specified in **response to line** or **char** definitions. See 14.2.2.3, "Match Clauses for Line or Character Input" on page 14-9 for more information on match clauses. When EASEL OS/2 EE finds a match in one of the **case** statements it then executes all of the action statements which are listed below the **case** statement.

No matter how many **case** statements match the string, only the first is executed. If EASEL OS/2 EE finds no **case** statements which match the STRING_VALUE it will do the action statements which follow the **default** statement (all **switch** statements must have a **default** statement and it cannot be followed by any other **case** statements).

```
switch (Appl_Var) is
    case char "bad transmission" is
    :
    case char "enter name" is
    :
    case char "more data" is
    :
    default is
    :
end switch
```

In the example if the value of the variable Appl_Var is the string "Please enter name:", the second **case** statement will be executed.

11.6 Program Timing

11.6.1 Responding to Elapsed Time

You specify the **response to interval** statement when you want one or more actions to occur every n seconds during the execution of an EASEL OS/2 EE program, or of any block within it.

Model

response to interval SECONDS

[ACTION_STATEMENT]...

In the following example, information in Status_Region is flashed on the screen every 60 seconds, and the time is updated:

```
response to interval 60
  read file Status into Status_Region
  change Clock to time
```

If other actions are taking place while the specified seconds have elapsed, **response to interval** will take place after the previous actions are completed. If the previous actions exceed the specified interval two or more times, **response to interval** will respond to the last interval only.

A **response to interval** statement can be specified within a block. The following example shows two **response to interval** statements, one inside Block_1 and one in the main program. When the program is in Block_1, only the **response to interval 15** is valid.

```
response to interval 60
:
response to Main_Menu
  begin guarded Block_1
    response to Restart
    enable Open_Menu
    leave block
    response to interval 15
    add to Timer
    pattern Graphics
  end Block_1
```

If Block_1 was *not* **guarded** and Main_Menu was selected, the **response to interval 15** would occur until 60 seconds passed; then the **response to interval 60** would occur, causing Block_1 to be exited.

11.6.2 Responding to Elapsed Time During Which No Stimulus is Received

You specify the **response to timeout** statement if you want an action to occur when EASEL OS/2 EE has not received any stimulus after a specified number of seconds.

Model

response to timeout SECONDS

[ACTION_STATEMENT]...

In the following example, if EASEL OS/2 EE has not received a stimulus from the user or from any application for 60 seconds, EASEL OS/2 EE will stop communications with Remote_Application and change to a new program.

```
response to timeout 60
    stop Remote_Application
    change to program Program_Logo
```

Note that **response to timeout** does not recognize **response to interval** as a stimulus. For example, if an EASEL OS/2 EE program has a message flashed on the screen every 15 seconds but does not receive any other stimulus for the specified number of seconds, the **response to timeout** statement will take effect.

A **response to timeout** statement can be specified within a block. The following example shows two **response to timeout** statements, one inside Block_1 and one in the main program. When the program is in Block_1, EASEL OS/2 EE will not execute the **response to timeout 60** statement in the main program, but will instead execute the **response to timeout 15** statement. This is because there can be only one active **response to timeout** statement at any given time. However, if the program is in the main program—or in any block which does not have a **response to timeout** statement—and nothing happens for 60 seconds, EASEL OS/2 EE will execute the **response to timeout 60** statement in the main program.

```

response to timeout 60
:
response to Main_Menu
  begin Block_1
    response to Restart
      enable Open_Menu
    response to timeout 15
      stop all
  end Block_1
:
  begin Block_2
:

```

11.6.3 Action Statement to Suspend Processing

The **wait** statement allows you to suspend processing of the EASEL OS/2 EE program. This is useful when you want to redraw the screen or when you are waiting for a remote application to perform a task unrelated to EASEL OS/2 EE.

When processing is suspended, EASEL OS/2 EE queues all stimuli including user input. When processing is resumed, the stimuli are processed in the order in which they were received.

All EASEL OS/2 EE processing is suspended until the specified number of seconds has elapsed. The maximum time is determined by the SLEEPMAX configuration option, which defaults to 60 seconds.

Model

wait SECONDS

In the following example, when EASEL OS/2 EE receives any input line from the Database application, it makes Chart visible and then waits approximately 10 seconds, queuing any stimuli that it might receive. After 10 seconds have elapsed, EASEL OS/2 EE processes any queued stimuli in the order in which they were received.

```

response to line from Database
  make Chart visible
  wait 10

```

The time that EASEL OS/2 EE waits is approximate. The actual elapsed time might be longer than the time you specify.

11.6.3.1 Using wait 0

Usually, EASEL OS/2 EE updates (redraws) the screen after the statement in a response definition has been executed. For example:

```
response to Go
  make object green
  make object red
  make object green
```

When EASEL OS/2 EE executes this response, the user will never see Go as red. But if you use the **wait 0** statement in between color changes, EASEL OS/2 EE will update the screen immediately, even though all the action statements in the response have not finished executing. Building on the above example, if you specify **wait 0** after the first two color changes:

```
response to Go
  make object green
  wait 0
  make object red
  wait 0
  make object green
```

the user will see Go flash green, red, and back to green.

11.7 Runtime Memory Handling

11.7.1 Global Memory

Global memory is used for several items. Many of these are relatively static, being allocated when the first program of an EASEL OS/2 EE interface is run and remaining fixed until the interface is terminated. These static items are:

I/O Control Blocks	Use approximately 1K of global memory, which is allocated at the beginning of the first EASEL OS/2 EE program to be run; it is never freed.
Pointing Device Buffers and Control Blocks	Use varies for each device; each device uses between 150 and 300 bytes. The memory is allocated at the beginning of the first EASEL OS/2 EE program to be run.
Application Buffers and Control Blocks	Use approximately 680 bytes for each application. They are allocated when an application is started, and de-allocated when it is stopped. These buffers may expand at runtime if EASEL OS/2 EE must store large amounts of data from applications.

11.7.1.1 Memory Restrictions in the Use of Global Data

The single most dynamic area that affects global memory is that of global variables. When the first EASEL OS/2 EE program is run there are no global variables in global memory. Working copies of global variables are actually maintained in program (local) memory.

When the first program exits via a **change to program** statement, all global variables are copied into global memory; when the next program is run, the values of the global variables are copied from global memory to local memory. They are not selected again until the program exits, at which time the values are again transferred to global memory. Thus, if the interface runs out of global memory, it will most likely do so only after a **change to program** statement has been executed.

Note: Arrays of global string variables are the most likely item to cause fragmentation of global memory, and should be avoided if you are going to use the **change to program** statement frequently.

11.7.2 Changing Memory Allocation in the EASEL.CMD File

11.7.2.1 Allocating Local Memory

During installation the PRGSIZE configuration option is set to 300, with a maximum size, PRGMAX, set to 500. This specifies that the amount of memory to be allocated to the program's .EBI file, along with any runtime expansion, will start at 300K and can grow to 500K.

If you want to use many (or large) local applications or non-EASEL OS/2 EE drivers, you may need to lower the PRGSIZE configuration to allow more memory for local applications. You can do this by editing the EASEL OS/2 EE CONFIG.ESL file or by invoking EASEL OS/2 EE with the **-m** option, which changes the PRGSIZE for this invocation only. For example:

```
easel sample -m 150000
```

The next time you run EASEL OS/2 EE, the memory will return to the value specified in the EASEL OS/2 EE configuration file.

11.7.2.2 Allocating Global Memory

During installation, the GLBSIZE configuration option is set to 10, with a maximum size, GLBMAX, set to 50. This specifies that the amount of memory to be allocated to store global data will start at 10K and can grow to 50K. This amount is sufficient for nearly all interfaces.

If you need to increase the amount of space allocated to global data, you can either edit the configuration file, CONFIG.ESL, and change the GLBSIZE entry to make the change permanent, or invoke EASEL OS/2 EE with the **-g** option. For example:

```
easel sample -g 15000
```

11.7.3 Controlling Memory from within the Program

Within an EASEL OS/2 EE program, you can determine how many bytes of memory are available for the program's use at any given time. If memory drops below a specified minimum, you can eliminate memory fragmentation, thus freeing more memory. The EASEL OS/2 EE statements used to handle memory are:

Statement	Purpose
freesize	Built-in function that returns the size of the largest available block of free memory, up to 64K.
response to low memory	Response statement that is automatically triggered when available memory has fallen below the specified threshold.

set low memory threshold	Action statement that specifies the minimum amount of memory required during the execution of the program. Determines when the response to low memory is triggered. (Should be less than 64K.)
squeeze memory	Action statement that eliminates fragmentation.

Generally, these statements are used in conjunction with each other. The **squeeze memory** action statement can take a few seconds to execute, so do not use it unless you have to. Each of these statements is described in detail below.

11.7.3.1 The freesize Function

The value returned by EASEL OS/2 EE for the **freesize** function is the byte size of the single largest available block of memory free to be used by the EASEL OS/2 EE program. This function can be used to monitor memory usage within your program.

Model

freesize

The following example illustrates how you might use the **freesize** function.

```

response to start
  set low memory threshold to 2000
  :
response to low memory
  squeeze memory
  if (freesize < 2000) then
    change to program Alternate_Choice
  end if

```

When **freesize** is used directly after the **squeeze memory** statement, as shown above, its value includes the sum of all available memory (since the **squeeze memory** statement creates one large block of free memory from all the fragmented pieces). Because an EASEL OS/2 EE program is made up of 64K blocks of memory, the largest value returned by **freesize** is 64K, even when more than 64K bytes are free.

11.7.3.2 The set low memory threshold Action Statement

The **set low memory threshold** statement designates the minimum amount of free memory (in bytes) needed during the execution of the program. This statement should be specified at the beginning of the program (usually in the **response to start** response statement). If it is not specified in the program, a default value of 5,000 bytes is set.

Model

set low memory threshold to BYTE_SIZE

The **set low memory threshold** statement allows you to set the trigger value for which the **response to low memory** response statement will be executed. For example, if you know that the minimum amount of free memory required in your program is 2000 bytes, you might specify:

```
response to start
    set low memory threshold to 2000
    :

response to low memory           # The response is
    change to program Next_Choice # executed if there
                                   # are less than 2000
                                   # bytes available.
```

The low memory threshold can be adjusted during the execution of the program. As shown in the following example, if EASEL OS/2 EE is adding text to a textual region and the low memory threshold is reached, the program can notify the user of the situation and allow the user to take actions until a minimum threshold is reached. The program will then write the contents of the textual region to an OS/2 EE file and exit gracefully.

```
integer Warn_Value is 10000
       Min_Value is 2000

response to start
    set low memory threshold to Warn_Value
    :
# Continues...

response to low memory
    squeeze memory
    if (freesize <= Warn_Value) then
        action ShowWarnMessage
        copy (Warn_Value - 5000) to Warn_Value
        if (Warn_Value < Min_Value) then
            write TextRegA to file "memreg.sav"
```

```

        exit
    else
        set low memory threshold to Warn_Value
    end if
end if

```

You should not set the low memory threshold above 64000.

11.7.3.3 The squeeze memory Action Statement

The **squeeze memory** action statement reorders the available memory in a program so that no fragmentation exists. This action takes a few seconds to execute, so do not use it unless you have to.

Model

squeeze memory

In the following example, the integer variable `SizeNeeded` contains the exact amount of memory needed. The boolean **true** will be copied to the variable `MemoryOk` if there is enough memory for an operation that would use `SizeNeeded` bytes. If there is not enough room, EASEL OS/2 EE will **squeeze memory** to try to create the needed memory. If the program still does not have enough memory after the squeeze, the boolean **false** will be copied to `MemoryOk`.

```

action CheckMemory is
    if (freesize >= SizeNeeded) then
        copy true to MemoryOK
    else
        squeeze memory
        if (freesize >= SizeNeeded) then
            copy true to MemoryOK
        else
            copy false to MemoryOK
        end if
    end if
end if

```

EASEL OS/2 EE automatically implements the **squeeze memory** function after you compile your program, and when you specify a **save program as** statement.

11.7.3.4 The response to low memory Response Statement

The **response to low memory** statement is taken when EASEL OS/2 EE reaches a minimum threshold of available memory during execution. The threshold is set either by you in the **set low memory threshold** action statement, or has a default value of 5,000 bytes.

There can be only one **response to low memory** statement in a program. It must be located in the outermost responses area of the program (that is, not inside a block structure).

Model

response to low memory

[ACTION_STATEMENT]...

During runtime, EASEL OS/2 EE processes an internal program memory check after execution of each response statement. The **response to low memory** statement is invoked when available memory has fallen below the low memory threshold (which is by default 5000 bytes, or set in the **set low memory threshold** statement). Before executing the response to low memory statement, EASEL OS/2 EE will attempt to increase available free memory by increasing the program memory size. If this fails, or if the amount of program memory has reached the maximum specified in the EASEL OS/2 EE configuration file, the response to low memory is invoked.

In the following example, if the threshold is reached during a response statement, the response statement will be completed, and the program will then execute the **response to low memory** and perform a **squeeze memory**. Then, if there is enough memory, EASEL OS/2 EE will return to the same state as before the **response to low memory** statement was taken. If there is still not enough memory, the action Warn_User takes place.

```
response to low memory
  squeeze memory
  if (freesize < 2000) then
    action Warn_User
  end if
```

The **response to low memory** statement will not be invoked if EASEL OS/2 EE is within a loop, since EASEL OS/2 EE checks memory only after the execution of a response statement, and a loop consists of a series of continuous action statements.

This response *will* interrupt a guarded block, but a guarded block cannot contain a **response to low memory** statement.

Chapter 12. Layout/CUA

12.1 Before You Begin

The Layout/CUA design tool allows you to create user interfaces that conform to IBM's Systems Application Architecture/Common User Access (SAA/CUA) guidelines. You can use Layout/CUA's What You See Is What You Get (WYSIWYG) editor to create CUA-conforming interfaces. You can convert these interfaces directly into EASEL OS/2 EE code that can be compiled and run using EASEL OS/2 EE. The Layout/CUA tool itself conforms to the CUA rules.

12.1.1 What You Need to Use Layout/CUA

Layout/CUA requires an IBM programmable workstation running OS/2 Extended Edition with a VGA or 8514/A graphics display adapter and a mouse. It requires 1Mb of available memory above the OS/2 Extended Edition requirements.

12.1.2 What You Need to Know

It is assumed that you are already familiar with using PM applications. It is also assumed that you are familiar with programming in EASEL OS/2 EE.

You should also understand the CUA standard for user interface design. You do not need to know specifics of the rules and exceptions; the Layout/CUA tool understands the CUA rules so that you do not have to.

12.1.3 About CUA Conformance

Layout/CUA allows you to create applications that use the Single Document Interface (SDI) style.

Layout/CUA does not allow you to create an interface that would break a CUA rule. If you break a rule, it displays a message informing you of the rule, and you cannot continue until you correct the error.

In some cases CUA defines exceptions to a rule. In these cases, Layout/CUA notifies you if you break the rule, notifies you of the exception, and allows you to take the exception.

12.1.4 About This Chapter

12.2, “Using Layout/CUA” on page 12-4 is a step-by-step tutorial for using Layout/CUA.

12.3, “Code Generation” on page 12-21, explains how the code generation options work.

12.4, “Object Information” on page 12-24, describes in detail the types of information associated with each Layout/CUA object.

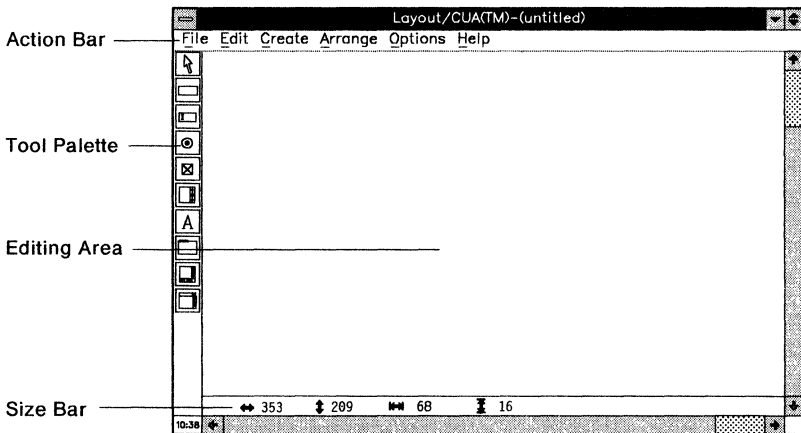
12.2 Using Layout/CUA

To run Layout/CUA, display the OS/2 command prompt and type:

```
es\cua
```

12.2.1 The Layout/CUA Primary Window

The Layout/CUA primary window comprises four major components, as shown in the following illustration:



Action bar

Use the action bar to read and write interfaces, access the clipboard, create windows, arrange controls, modify object information, and get help.

Tool palette

Use the tool palette to add controls to a dialog box or dialog region. You can also use the selection tool from the palette to select, move, and resize objects.

Size bar

Turn on the size bar option to display the size and position of the currently selected control.

Editing area

The central portion of the primary window is the editing area, where you build your interfaces. You can use the scroll bars, which are on the bottom and right side of the primary window, to scroll the editing area.

12.2.2 Using the Help System

The built-in help system is one of the most important parts of the Layout/CUA tool. It contains most of the information in this chapter. Therefore, if you understand how to use the help system, you will not need to constantly refer to the chapter.

You can access the help system in a number of ways.

12.2.2.1 General Help

Press F1 for help. If the help system does not have specific information for the current action, it displays general information on Layout/CUA.

You can also select a choice from the Help pulldown for general help, as follows:

- Extended Help... for general information about Layout/CUA.
- Help Index... to display an index of topics contained in the help system. To get help on a topic, move the mouse to the topic and double click.
- Keys Help... to display all the actions you can use the keyboard for.
- About... to display the version number and the message usage. The memory indicator shows the remaining memory available for building a user interface.

12.2.2.2 Action Bar Help

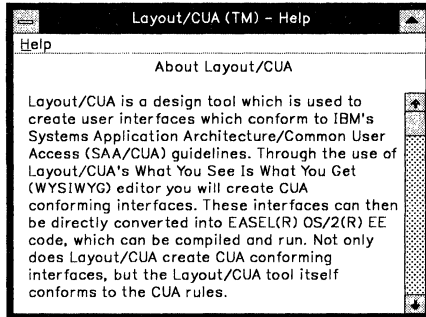
Use the mouse to select an item from the action bar, then, while holding down the left mouse button, move the mouse until the choice you want help on is highlighted. Press F1 to display help information for the highlighted choice.

12.2.2.3 Dialog Box Help

Press the Help push button or press F1 to display help information for Layout/CUA dialog boxes.

12.2.2.4 The Help Window

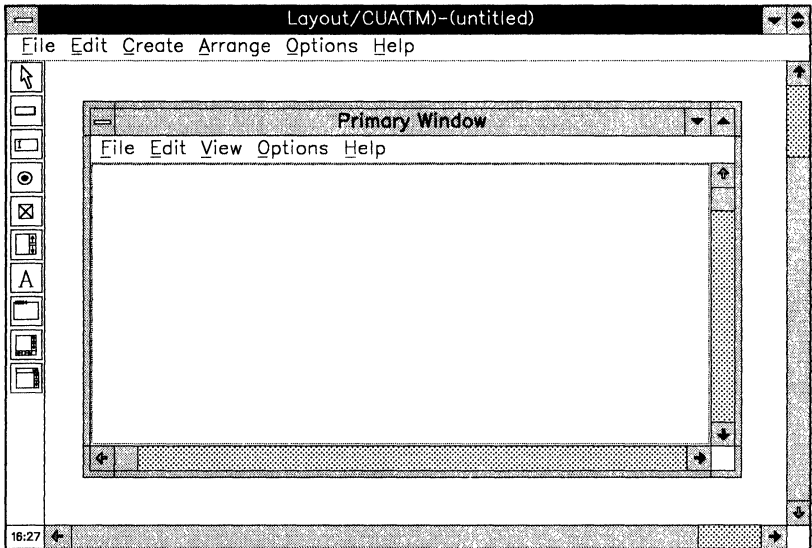
Help information is displayed in a help window, as shown below. The help window has the same Help pulldown as displayed in the Layout/CUA action bar. You can move, resize, or maximize the help window. You can continue to work with Layout/CUA while the help window is displayed.



12.2.3 Creating an Interface Primary Window

When you start the Layout/CUA application, the editing area is empty. You can either create a new interface or edit an existing one.

Select New from the File pulldown if you want to create a new interface. This creates the primary window for your interface, as shown below. An interface can have only one primary window.



A primary window can be an EASEL OS/2 EE:

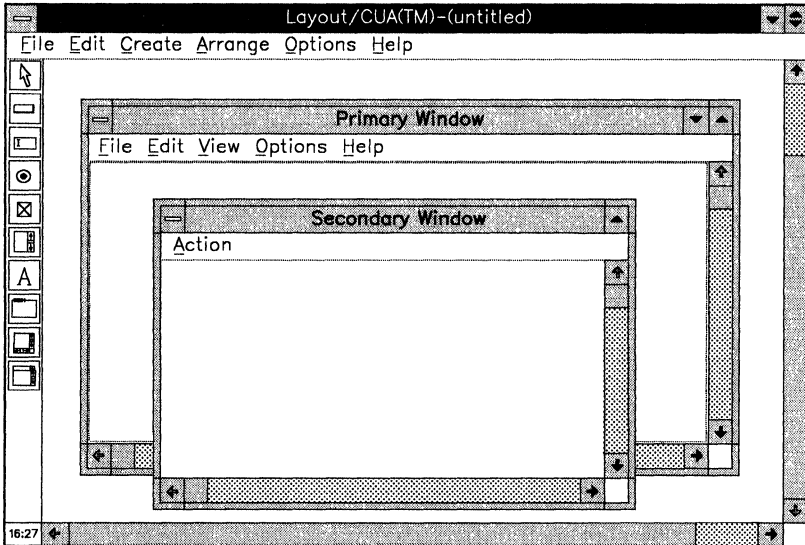
- Graphical region
- Dialog region
- Textual region
- Image region

12.2.3.1 Adding Objects

The type of object you want to add determines how you do so. Use pulldown choices to add windows; the tool palette to add controls.

12.2.4 Creating Secondary Windows

Select Secondary Window from the Create pulldown to create a secondary window, as shown below. You can add up to 20 secondary windows to an interface created with Layout/CUA.

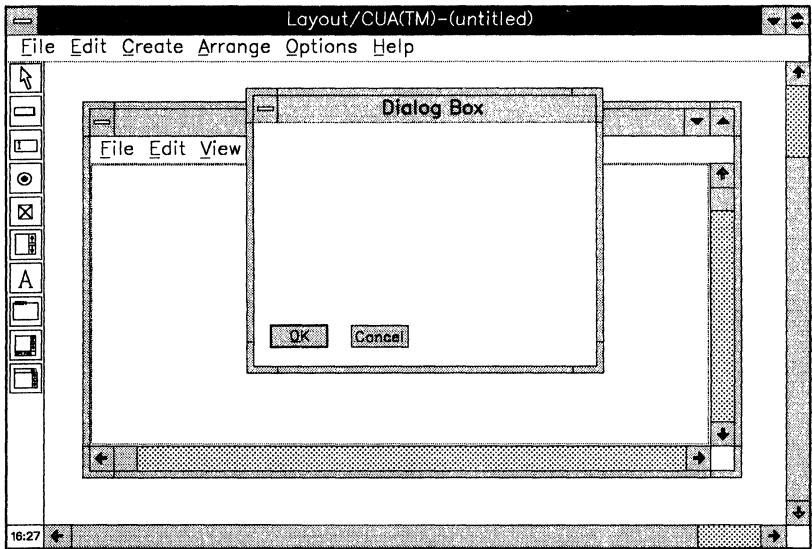


A secondary window can be an EASEL OS/2 EE:

- Graphical region
- Dialog region
- Textual region
- Image region

12.2.5 Creating Dialog Boxes

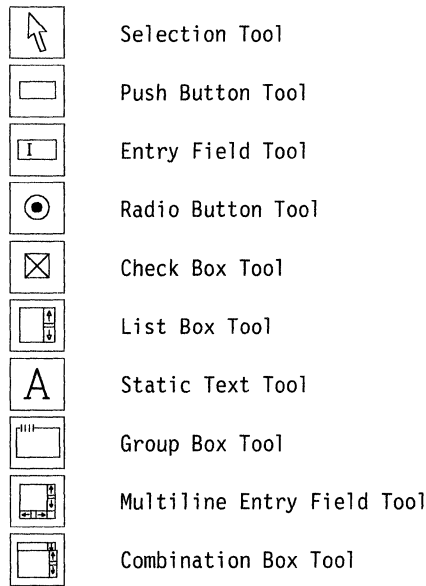
Select Dialog Box from the Create pulldown to create a dialog box, as shown below. This automatically creates a modal dialog box, with the OK and Cancel buttons already added. You can add up to 30 dialog boxes to an interface created with Layout/CUA.



12.2.6 Selecting a Tool from the Palette

Click on a tool from the palette to select that tool. When you select a tool, the mouse cursor changes to the shape of that tool, with a small cross-hair in the lower-left corner. You can then add the control associated with that tool to the dialog box or dialog region.

The tools are shown in the following figure.



Note that the selection tool, which is the first tool in the palette, does not create a control. You use it to select objects.

When you choose a tool, the left and right mouse buttons have different functions, as follows:

Mouse Button	Use
Left	Permanent tool selection. For adding multiple controls of the same type. Changes the mouse cursor to that tool, until you select another tool.
Right	Temporary tool selection. For adding one control of a selected type. Changes the mouse cursor to that tool temporarily. After the control is added, the cursor changes back to the selection tool.

12.2.7 Adding Controls to a Dialog Box or Dialog Region

After selecting a tool, move the cursor to the place in the dialog box or dialog region where you want to add the control. The cross-hair shows the exact position at which the lower left-corner of the new control will be added. Press either mouse button to add the control. The control's Info dialog box is automatically displayed. (Info dialog boxes are discussed in 12.4, "Object Information" on page 12-24.) To accept the information in the dialog box, select OK.

12.2.8 Selecting Controls Inside a Dialog Box or Dialog Region

You use the selection tool to select a control. If the cursor is not in the shape of an arrow, click on the selection tool (the first tool in the palette). Then click inside the control you want to select. A red "marching-ants" pattern identifies the currently selected control.

You can select more than one control by moving the mouse to another control, holding down SHIFT, and clicking. SHIFT/click inside a control to deselect it.

You can delete, move, resize, edit, and cut/copy and paste a select object. You can delete, move, resize, align, and cut/copy and paste groups of objects.

The left and right mouse buttons have different functions when used to select a control, as follows:

Mouse Button	Use
Left	To select a control and optionally move it. Use if you want to move the selected control.
Right	To select a control without moving it. Use if you want to make sure you do not accidentally move the selected control.

If you had used the permanent tool selection method (clicking with the left button) to create controls, then you would need to click on the selection tool before selecting a control. If you had used the temporary tool selection method (clicking with the right button) to create a single control, then the selection tool would be selected automatically after the control was created. You can always determine which tool you have by looking at the shape of the cursor.

12.2.9 Deleting Objects

Select the object you want to delete. Then select Clear from the Edit pulldown to remove the object from the interface.

When you delete a dialog box or dialog region, all controls associated with the dialog box or dialog region are also deleted.

12.2.10 Moving Objects

You use the selection tool to move an object. The type of object you want to move determines how you do so.

To move a window:

1. Move the selection cursor to the window's title bar.
2. Press and hold the left or right mouse button.
3. Move the window to the new location. An outline of the window moves as you move the mouse.
4. Release the button. The window appears in the new location.

To move a control:

1. Select the control.
2. Make sure the cursor is a single arrow, and then press and hold the left mouse button.
3. Move the control to the new location. The entire control moves as you move the mouse.
4. Release the button.

To move a group of two or more objects:

1. Select the objects in the group (using SHIFT/click, described previously).
2. Move the cursor to one of the objects in the group, and make sure the cursor is a single arrow.
3. Press and hold the left mouse button.
4. Move the entire group to the new location.
5. Release the button.

12.2.11 Resizing Objects and Groups

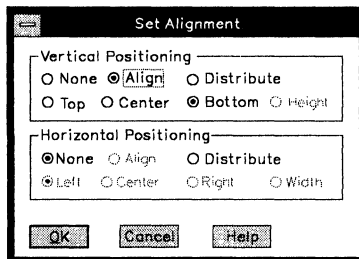
You use the selection tool to resize an object or group, as follows:

1. Select the window, control, or group.
2. Move the cursor to the side or corner you want to resize, and make sure the cursor is a double-pointed arrow.
3. Press and hold the left mouse button.
4. Move the side or corner until the object or group is the desired size.
5. Release the button.

12.2.12 Aligning Objects

12.2.12.1 Specifying Alignment Options

Select Set Alignment... from the Arrange pulldown to display the Set Alignment dialog box, as shown below. Use the box to specify the type of alignment you want.



You can align groups of objects vertically or horizontally. Select None from the Vertical Positioning or Horizontal Positioning box to specify no alignment in that direction. Select Align to align objects vertically or horizontally. Select Distribute to distribute objects vertically or horizontally.

You can specify the direction of alignment. For example, select Left to align the left edges of the objects with the leftmost object in the group.

Select Distribute and a dimension to distribute objects evenly by height or by width within the dialog box.

12.2.12.2 Aligning a Group of Objects

Select Align from the Arrange pulldown to align a group of objects according to the current settings of the Set Alignment dialog box. Align is available only when you select two or more objects.

12.2.13 Modifying Object Information

Each Layout/CUA object has information relating to its EASEL OS/2 EE name and attributes that can change its characteristics. Use one of the following methods to display an Info dialog box that allows you to modify this information:

- Choose the selection tool, and then double-click on the object whose information you want to modify.
- Use the selection tool to select the object whose information you want to modify. Then select Info from the Options action bar pulldown. (You can use the accelerator key, Ctrl-I, instead of selecting from the pulldown.)

You can then edit and save the information in the dialog box. See 12.4, “Object Information” on page 12-24 for details of the information associated with each object type.

12.2.14 Editing the Action Bar

Select the action bar item you want to edit. Select Edit Action Bar from the pulldown to display the Action Bar Choice Info dialog box. This allows you to edit or delete the action bar item, or add a new action bar item after the current one.

To add a new item to the action bar, change the text and object name fields to what you want the new item to read, and click on the Insert push button.

12.2.15 Editing a Pulldown

Select the action bar item that contains the pulldown you want to edit. Select a choice from the pulldown to display the Pulldown Choice Info dialog box. This allows you to edit or delete the choice or add a new choice after the current one.

To add a new choice to the pulldown, change at least the text and object name fields to what you want the new choice to read, and click on the Insert push button.

12.2.16 Undoing Your Last Action

Select Undo from the Edit pulldown to undo your last action. The text of the Undo choice varies, depending on the last action. For example, if the last thing that you did was create a window, the Undo choice displays Undo Create, as shown below.

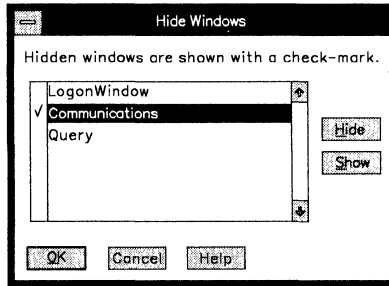
Edit	
Undo Create	Alt+Backspace
Cut	Shift+Del
Copy	Ctrl+Ins
Paste	Shift+Ins
Clear	Del

12.2.17 Hiding and Showing Windows

Creating a complex application with many secondary windows and dialog boxes can clutter the editing area so that it becomes difficult to work with. You can alleviate this problem by temporarily hiding some of the windows. The windows are not removed from the interface; they are simply hidden from view.

Use the following procedure to hide a window:

1. Select Hide Windows... from the Arrange pulldown to display the Hide Windows dialog box, as shown below.



2. Select from the list box the window or windows you want to hide.
3. Select the Hide push button.
4. Select the OK push button.

Use the following procedure to reveal a hidden window:

1. Select Hide Windows... from the Arrange pulldown to display the Hide Windows dialog box.
2. Select from the list box the window or windows you want to reveal.
3. Select the Show push button.
4. Select the OK push button.

12.2.18 Creating Selection Fields

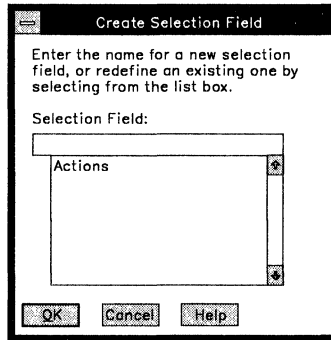
If you are using radio buttons, you must group them into a selection field to specify which radio buttons are associated with each other, because only one radio button in a group can be selected at a time. You can also group push buttons and check boxes into selection fields.

You can have only one type of control in a selection field; for example, you cannot create a selection field with radio buttons and push buttons, or push buttons and check boxes. A control can be in one selection field only.

Use the following procedure to create a selection field:

1. Select the controls you want to include in the selection field. Remember that you can select several objects by holding down SHIFT and clicking on the second and subsequent objects.

2. Select Selection Field... from the Create pulldown to display the Create Selection Field dialog box, as shown below.



3. Fill in the name for the selection field.
4. Click the OK push button.

12.2.19 Using the Clipboard

You can cut or copy an object into the system clipboard and then paste it back into the interface in a different place. You can also paste it into any other application that understands the format of the data. Layout/CUA places data in the clipboard in two formats:

- An internal format that only Layout/CUA understands
- The standard PM bitmap format

This means that you can paste objects from your interface into applications such as paint programs or desktop publishing systems. If you place a dialog box in the clipboard, all its controls are also placed in the clipboard. The clipboard can also be very handy when creating documentation for your application.

12.2.19.1 Cutting and Copying an Object

You can cut an object from the interface and place it in the clipboard, or copy it to the clipboard without removing the original from the application.

Use the following procedure to cut or copy an object to the clipboard:

1. Select the object you want to cut or copy.
2. Select Cut or Copy from the Edit pulldown.

The object is then in the clipboard and available for pasting.

Note that code generation options and selection fields are not carried with the object when it is placed in the clipboard.

12.2.19.2 Pasting an Object

Once an object is in the clipboard, you can paste it back into the interface. If the object in the clipboard is a control, you can paste it into a dialog box or a dialog region.

Use the following procedure to paste a control from the clipboard in a dialog box:

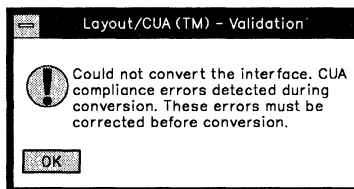
1. Select the dialog box you want to paste the control into.
2. Select Paste from the Edit pulldown.

The text of the Paste choice varies, depending on what is currently in the clipboard. For example, if there is a control in the clipboard the Edit pulldown displays Paste Control.

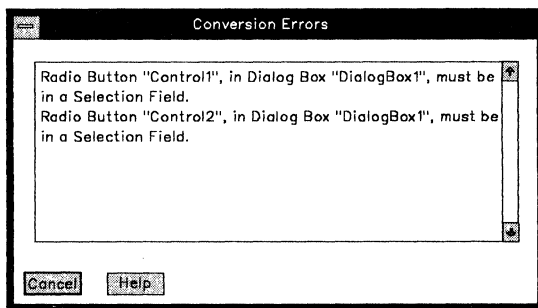
12.2.20 Converting the Interface to EASEL OS/2 EE Code

Once your interface is complete, you can convert it into EASEL OS/2 EE code, using the following procedure:

1. Select Convert... from the File pulldown. The screen that appears next depends on whether your interface has CUA errors.
 - If there are CUA errors, a message box appears, as shown in the following figure.

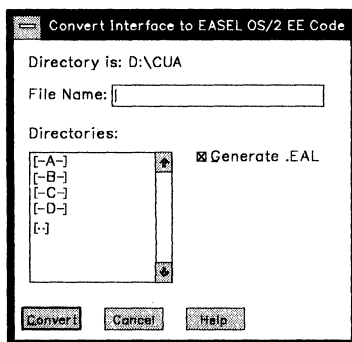


Select OK in the message box to display the Conversion Errors dialog box, as shown below.



You must correct the errors before the interface can be converted. However, you can save the interface to a file without correcting the errors, since saving an interface does not generate EASEL OS/2 EE code. (See 12.2.21, “Saving the Interface” on page 12-19.) You can then retrieve and correct the interface at a later time. Select Conversion Errors from the File pulldown at any time to display the Conversion Errors dialog box again.

- If there are no CUA errors, the Convert dialog box appears, as shown below.



2. Click in the Directories list box if you want to change the directory or drive. You can save the converted interface to any directory.
3. Enter the name of the file that you want the EASEL OS/2 EE code written to.

When the interface is converted, Layout/CUA will create two files, and will assign the extensions as follows:

Main source code file: .EAL
 Objects include file: .INC

The main source code file (.EAL) contains everything except the object definitions (environment declarations, action routine definitions, response definitions, etc.). The object include file (.INC) contains all the object definitions.

4. Use the Generate .EAL check box to indicate whether or not you want to output the main source code file (.EAL). The object include file will always be generated. The first time you convert an interface, both the .EAL and .INC files should be created. If you then change an object definition through Layout/CUA, you can output only the .INC file, without overwriting the .EAL file. If you have modified the .EAL file to add your own code, you will not lose your changes.
5. Select the Convert push button to begin conversion. Layout/CUA makes a final check for CUA conformance and displays a message if the interface still has CUA errors. Otherwise, it converts the interface.

The generated EASEL OS/2 EE can now be compiled and executed or updated to include the additional EASEL OS/2 EE language statements necessary to complete the application.

12.2.21 Saving the Interface

You can save the interface to an existing file or to a new file. The interface is saved in a format that only Layout/CUA can read. This is useful if you want to be able to edit or enhance the interface later.

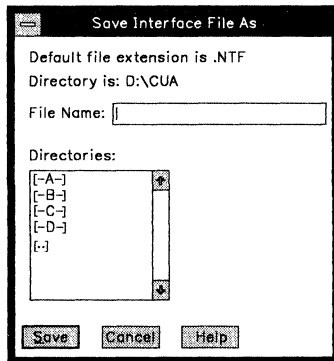
Note that saving an interface does not generate EASEL OS/2 EE code.

12.2.21.1 Saving to an Existing File

Select Save from the File pulldown to save the interface to an existing file. Layout/CUA immediately writes the new interface over the previous one, without displaying a dialog box.

12.2.21.2 Saving to a New File

Select Save As... from the File pulldown to save the interface to a new file. Layout/CUA displays the Save Interface File As dialog box, as shown below.



Fill in the filename. You do not need to supply an extension; Layout/CUA supplies the extension .NTF, by default. Select Save to save the new file.

12.2.22 Editing an Interface File

Select Open from the File pulldown to open an existing interface file for editing. Layout/CUA displays a dialog box that allows you to specify which interface file to open. Select the filename you want to open and then select Open.

12.3 Code Generation

When an interface is converted to EASEL OS/2 EE code, a number of things take place, in the following order:

1. The objects from the interface are converted to their representation in EASEL OS/2 EE code.
2. Subroutines are generated to access the created dialog boxes.
3. Responses are added for any pulldown choices that are to bring up a dialog box.

The following describes the subroutines that Layout/CUA creates when it converts an interface to EASEL OS/2 EE code.

12.3.1 Dialog Box Subroutines

If your interface has modal dialog boxes defined within it, a set of subroutines will be generated that you can use to perform dialog box management. These subroutines make it easier to perform the normal actions that need to be done, to make dialog boxes behave correctly.

12.3.1.1 Setup Subroutine

The setup subroutine is used to initialize the contents of the dialog box. For every control in the dialog box that uses the code generation option, there will be a parameter in the parameter list of the subroutine. For example: if a check box is using code generation, you can pass a boolean variable as a parameter, and the variable's setting will automatically check or uncheck the control. The setup subroutine should be called before the dialog box is displayed.

```
subroutine SetValueDialog( boolean : OnOff ) is
  if ( OnOff ) then
    check Setting in ValueDialog
  else
    uncheck Setting in ValueDialog
  end if
```

If the dialog box has no controls that are using the code generation option, the setup subroutine is not generated.

12.3.1.2 Process Subroutine

The process subroutine is used to display the dialog box. It contains responses for any push buttons that are defined as displaying a dialog box. Responses are also generated for the OK and Cancel push buttons. This subroutine does not return until the user selects the OK or Cancel push button.

The subroutine has only one parameter: DidOKCUA. This is a boolean variable that is set to true if the user selects OK; false if the user selects Cancel. This variable should be used to determine if you need to process the dialog box further, or do nothing because the user selected Cancel.

The following is an example of the process subroutine for the ValueDialog dialog box.

```
subroutine ProcessValueDialog( boolean : DidOKCUA ) is
  begin guarded
    response to start
      make ValueDialog visible
    response to OK in ValueDialog
      copy true to DidOKCUA
      make ValueDialog invisible
      leave block
    response to Cancel in ValueDialog
      copy false to DidOKCUA
      make ValueDialog invisible
      leave block
  end
```

This routine can be modified to create special handling of a dialog box.

12.3.1.3 Query Subroutine

The query subroutine is used to retrieve the data from the dialog box into variables. For every dialog box control that uses the code generation option, there will be a parameter in the parameter list of the subroutine. By calling the subroutine with a parameter list of variables, you can directly retrieve the contents of the dialog box without having to use the built-in functions.

The query subroutine is normally called when the user selects the OK push button.

```
subroutine QueryValueDialog( boolean : OnOff ) is
  copy ( Setting in ValueDialog is checked ) to OnOff
```

If the dialog box has no controls that are using the code generation option, the query subroutine is not generated.

12.3.2 Automatic Dialog-Box Calling

Layout/CUA can also generate the code to call dialog box routines when a pulldown choice or a push button in a dialog box is selected.

12.3.3 Pulldown Choices

If the code generation option to display a dialog box is selected for a pulldown choice, a response will be created that calls the dialog box sub-routines. The setup and query subroutines will be commented out since it is up to you to fill in the parameter list for these calls. Since the process sub-routine call is not commented out, it allows you to test the application without any further coding.

```
response to item Setup from SecondaryWindow1ABCUA
  #call SetupSetupDialog( ParityOnOff )
  call ProcessSetupDialog( DidOKCUA )
  #if ( DidOKCUA ) then
    #call QuerySetupDialog( ParityOnOff )
  #end if
```

12.3.4 Push Buttons

The code generation option to display a dialog box from a push button will create a response in the same manner as for pulldown choices. This response will be placed in the process subroutine associated with the push button's dialog box.

```
subroutine ProcessCallingDialog( boolean : DidOKCUA ) is
  begin guarded
    response to start
      make CallingDialog visible
    response to Control1 in CallingDialog
      copy true to DidOKCUA
      make CallingDialog invisible
      leave block
    response to Control2 in CallingDialog
      copy false to DidOKCUA
      make CallingDialog invisible
      leave block
    response to Values in CallingDialog
      #call SetValueDialog( OnOff )
      call ProcessValueDialog( DidOKCUA )
      #if ( DidOKCUA ) then
        #call QueryValueDialog( OnOff )
      #end if
  end
end
```

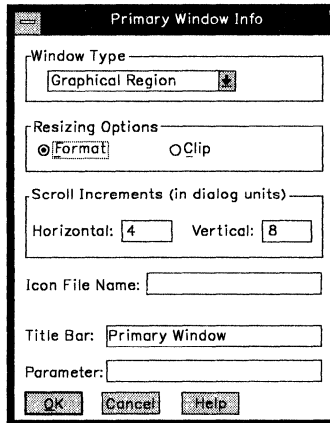
12.4 Object Information

This section contains detailed descriptions of the information associated with each Layout/CUA application object. Each object has an Info dialog box associated with it. This dialog box is used to modify the information, and in the case of the action bar, to insert and delete choices.

All objects have an EASEL OS/2 EE object name associated with them. Many also have a parameter (this is the EASEL OS/2 EE object parameter).

12.4.1 Primary Window Info

The Primary Window Info dialog box, shown below, contains the settings for window type, resizing, scrolling, and minimizing.

The image shows a dialog box titled "Primary Window Info". It contains several sections: "Window Type" with a dropdown menu showing "Graphical Region" and a small icon; "Resizing Options" with two radio buttons, "Format" (selected) and "Clip"; "Scroll Increments (in dialog units)" with two input fields, "Horizontal" (value 4) and "Vertical" (value 8); "Icon File Name:" with an empty text field; "Title Bar:" with a text field containing "Primary Window"; "Parameter:" with an empty text field; and at the bottom, three buttons: "OK", "Cancel", and "Help".

The window type setting allows you to specify what type of EASEL OS/2 EE region the primary window will be created as. If the window is a dialog region, controls can be placed directly into it. (Code generation is not supported for dialog regions.)

The resizing options allow you to specify whether the window will format or clip when resized. Specifying Format means that when the window is resized, the contents will be scaled up or down. Specifying Clip means that when the window is resized, the contents will not change, and will be clipped to the edges of the window.

The scroll increment values are used to specify the amount the window will scroll when the horizontal and vertical scroll bars are used. Note that the values are specified in dialog units. Four dialog units is the average char-

acter width. Eight dialog units is the average character height. These are the default settings.

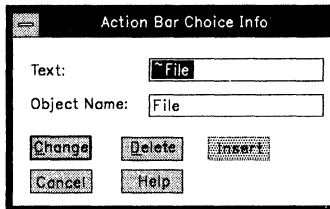
The Icon File Name entry field is used to specify the file that contains the icon to be displayed when the application is minimized. If the field is left blank, the default PM minimize icon will be used. The specified icon file does not have to exist at the time the interface is being created.

The Title Bar entry field contains the text that will be displayed in the primary window's title bar.

The EASEL OS/2 EE object name for the primary window is automatically generated by Layout/CUA as "Primary Window" and cannot be changed. Therefore, it is not displayed.

12.4.2 Action Bar Choice Info

The Action Bar Choice Info dialog box, shown below, allows you modify or delete the current action bar choice. It also allows you insert a new action bar choice following the current one.



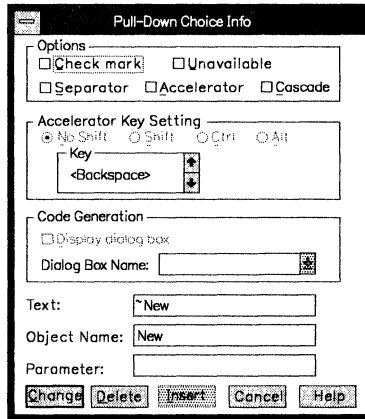
The Text entry field contains the text for the choice. The tilde (~) character should be placed before the character which is to be used as the mnemonic for the choice.

Selecting the Delete push button will delete this choice from the action bar. It will also delete its associated pulldown. Once the information in the dialog box has been changed, the choice cannot be deleted.

A new action bar choice can be added by filling new information for the text and object name, and selecting the Insert push button. Insert is only available after both the text and object name fields have been changed.

12.4.3 Pulldown Choice Info

The Pulldown Choice Info dialog box, shown below, allows you to specify the visual characteristics of the choice, as well as the accelerator key, and code generation options.



Checking the Check Mark check box will place a check mark in front of the choice.

The Unavailable check box is used to specify whether the initial state of the choice is enabled or disabled. When the choice is made unavailable (disabled) it will not be grayed while it is displayed in a Layout/CUA interface. When converted to EASEL OS/2 EE code and run, the choice will be grayed.

The Separator Line check box can be used to add a separator line after the choice.

Checking the Accelerator Key check box specifies that the choice is to have an accelerator key associated with it.

Checking Cascade specifies that the choice is to have a cascading pulldown. The choice will appear with an arrow to the right of the text. Only one level of cascading is allowed.

The Accelerator Key Setting options allow you to specify the accelerator key for the choice. You can use the scroll arrows to scroll through the list of valid keys. The options can be used to specify that the key is to be used shifted, unshifted, with control, or with Alt. Not all options are available for all keys; this is done to conform to CUA rules.

The code generation option for a pulldown choice is used to specify that selecting this choice is to display a dialog box. If this option is selected you must specify the dialog box name by selecting the down arrow to the right of the Dialog Box Name entry field. This will display a list box of available dialog boxes. You can select from this list by double-clicking. In order to hook a dialog box to a pulldown choice the dialog box must have already been created.

The Text entry field contains the text which will be displayed in the choice. The tilde (~) character should be placed before the character that will be used as the mnemonic for the choice. If the choice has an accelerator key, you do not need to add the text for the accelerator key. It will be done automatically for you at conversion time.

Selecting the Delete push button will delete this choice from the pulldown. Once the information in the dialog box has been changed, the choice cannot be deleted.

A new pulldown choice can be added by filling in new information for the choice, and selecting the Insert push button. Insert is only available after both the text and object name fields have been changed.

12.4.4 Secondary Window Info

The Secondary Window Info dialog box, shown below, is very similar to the Primary Window Info dialog box.

The image shows a screenshot of the 'Secondary Window Info' dialog box. It has a title bar with the text 'Secondary Window Info'. Inside the dialog, there are several sections: 'Window Type' with a pulldown menu showing 'Graphical Region'; 'Options' with two checkboxes, 'Visible' (checked) and 'Action Bar' (checked); 'Resizing Options' with two radio buttons, 'Format' and 'Clip' (selected); 'Scroll Increments (in dialog units)' with two input fields, 'Horizontal' (4) and 'Vertical' (8); 'Title Bar' with an input field containing 'Secondary Window'; 'Object Name' with an input field containing 'Secondary Window1'; and 'Parameter' with an empty input field. At the bottom are three buttons: 'OK', 'Cancel', and 'Help'.

The Window Type setting allows you to specify what type of EASEL OS/2 EE region the secondary window will be created as. If the window is a dialog region, controls can be placed directly into it. (Code generation is not supported for dialog regions.)

The Visible check box is used to specify whether the secondary window should be initially displayed visible. Even though the secondary window may be defined as invisible, it will be visible in the Layout/CUA interface. When the interface is converted to EASEL OS/2 EE code the visibility will be generated correctly.

The Action Bar check box is used to specify whether the secondary window is to have an action bar.

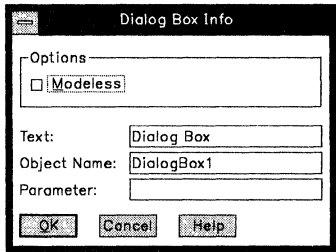
The Resizing Options allow you to specify whether the window will format or clip when resized. Specifying Format means that when the window is resized, the contents will be scaled up or down. Specifying Clip means that when the window is resized, the contents will not change, and will be clipped to the edges of the window.

The Scroll Increment values are used to specify the amount the window will scroll when the horizontal and vertical scroll bars are used. Note that the values are specified in dialog units. Four dialog units is the average character width; eight dialog units is the average character height. These are the default settings.

The Title Bar entry field contains the text to be displayed in the secondary window's title bar.

12.4.5 Dialog Box Info

The Modeless check box of the Dialog Box Info dialog box, shown below, is used to specify whether the dialog box is to be modal or modeless.



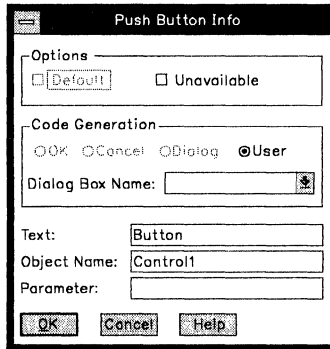
The code generation options for controls in the dialog box are not available for a modeless dialog box. This also means that a modal dialog box cannot be changed to modeless if any of its controls are using code generation.

When a new dialog box is created, it is created as a modal dialog box. Since the automatically added OK and Cancel push buttons are created with the OK and Cancel code generation options, this must be changed before changing the modality. You must change the code generation options for the push buttons to User.

The Title Bar entry field contains the text to be displayed in the dialog box's title bar.

12.4.6 Push Button Info

The Push Button Info dialog box, shown below, allows you to change the visual characteristics of the push button as well as its code generation options.



The Default check box is used to specify whether this is the Default push button. The Default push button is distinguished from other push buttons by its thick border. When running the dialog box, the action associated with the Default push button can be taken by pressing ENTER. If a Default push button has already been defined for the dialog box, the check box will be grayed.

The Unavailable check box is used to specify whether the initial state of the push button is enabled or disabled. When the push button is made unavailable (disabled), the push button will not be grayed while it is displayed in a Layout/CUA interface. When converted to EASEL OS/2 EE code and run, the push button will be grayed.

The Code Generation options for a push button are OK, Cancel, Dialog, and User.

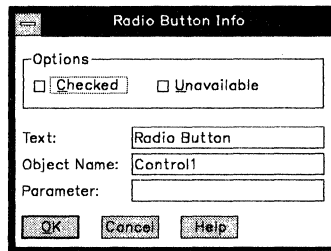
- OK and Cancel are used to specify the OK and Cancel push buttons for the dialog. Only one OK and one Cancel push button is allowed per dialog box. They will be grayed if already used by another push button.
- The Dialog option is used to specify that the push button will display another dialog box. If this option is selected, you must specify the dialog box name by selecting the down arrow to the right of the Dialog Box Name entry field. This will display a list box of available dialog boxes. You can select from this list by double-clicking. In order to hook a dialog box to a push button, the dialog box must have already been created.

- The User option is used to specify that the action associated with the push button is user defined. This specifies no code generation for the action associated with this push button.

The Text entry field contains the text to be displayed in the center of the push button. The tilde (~) character should be placed before the character to be used as the mnemonic for the push button.

12.4.7 Radio Button Info

The Checked check box of the Radio Button Info dialog box, shown below, is used to specify whether the initial state of the radio button is to be checked.



The Unavailable check box is used to specify whether the initial state of the radio button is enabled or disabled. When the radio button is made unavailable (disabled), it will not be grayed while it is displayed in a Layout/CUA interface. When converted to EASEL OS/2 EE code and run, the radio button will be grayed.

The Text entry field contains the text to be displayed to the right of the radio button. The tilde (~) character should be placed before the character to be used as the mnemonic for the radio button.

The code generation option for radio buttons allows you to pass the object name of a radio button to the setup subroutine. This will automatically check the specified radio button. The query subroutine will return the object name of the currently selected radio button of the selection field. The code generation option for radio buttons is not specified for each radio button, but for the selection field to which a group of radio buttons belong. It is accessed through the Manage Selection Fields choice in the Options pulldown.

12.4.8 Check Box Info

The Checked check box of the Check Box Info dialog box, shown below, is used to specify whether the initial state of the check box is to be checked.

Check Box Info

Options

☒ Checked ☐ Unavailable

Code Generation

☐ Yes Variable:

Text:

Object Name:

Parameter:

OK Cancel Help

The Unavailable check box is used to specify whether the initial state of the check box is enabled or disabled. When the check box is made unavailable (disabled), the check box will not be grayed while it is displayed in a Layout/CUA interface. When converted to EASEL OS/2 EE code and run, the check box will be grayed.

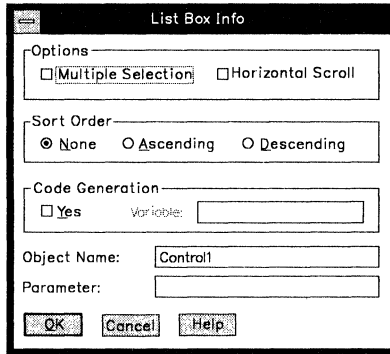
The code generation option for check boxes will generate code for the setup and query dialog subroutine that will manage the check box. The Variable entry field should contain the name of the boolean variable to be used in the parameter list of the dialog box subroutines.

When calling the subroutines, a boolean variable should be passed for this parameter. The setup subroutine will check or uncheck the check box dependent on the setting of the variable. The query subroutine will return the current state of the check box in the variable as true or false. True signifies checked, and false signifies unchecked.

The Text entry field contains the text to be displayed to the right of the check box. The tilde (~) character should be placed before the character to be used as the mnemonic for the check box.

12.4.9 List Box Info

The List Box Info dialog box, shown below, allows you to specify the type of list box, sort order, and code generation options.



The screenshot shows a dialog box titled "List Box Info". It contains three main sections: "Options", "Sort Order", and "Code Generation". In the "Options" section, there are two checkboxes: "Multiple Selection" (which is checked) and "Horizontal Scroll" (which is unchecked). In the "Sort Order" section, there are three radio buttons: "None" (which is selected), "Ascending", and "Descending". In the "Code Generation" section, there is a checkbox labeled "Yes" (which is unchecked) and a text field labeled "Variable:" which is empty. Below these sections, there are two text fields: "Object Name:" with the value "Control" and "Parameter:" which is empty. At the bottom of the dialog box, there are three buttons: "OK", "Cancel", and "Help".

The Multiple Selection check box is used to specify whether the list box is to be a multiple selection list box or not. Currently, code generation is not supported for multiple selection list boxes.

The Horizontal Scroll check box is used to specify whether the list box is to have a horizontal scroll bar. This can be used when the entries to be displayed in the list box are wider than the list box itself.

The Sort Order can be specified as None, Ascending, or Descending. If None is specified, list box entries are added according to the order of insertion. If Ascending or Descending is selected, the list box will automatically sort itself independent of the order of insertion.

The Code Generation option for list boxes will generate code for the setup and query dialog subroutine that will manage the list box. The Variable entry field should contain the name of an integer variable to be used in the parameter list of the dialog box subroutines.

When calling the subroutines, an integer variable should be passed for this parameter. The setup subroutine will select an entry in the list box, depending on the setting of the variable. The query subroutine will return the currently selected entry of the list box. The entry of the list box is represented in the integer variable as the line number of the entry. For example, the fifth entry in the list box is represented by the value five in the variable.

12.4.10 Entry Field Info

The Entry Field Info dialog box, shown below, is used to specify the characteristics of the entry field, as well as its Code Generation options.

Entry Field Info

Options

☒ Unavailable Text Size: 30

☐ Password

Code Generation

☐ Yes Variable:

Text:

Object Name: Control2

Parameter:

OK Cancel Help

The Unavailable check box is used to specify whether the initial state of the entry field is to be enabled or disabled. When an entry field is unavailable, a user cannot enter text into it. When the entry field is made unavailable (disabled), the entry field is not grayed when displayed in a Layout/CUA interface.

The Password check box is used to specify whether the entry field is to be a password entry field. Password entry fields display characters as asterisks to hide the actual information entered.

The Text Size entry field specifies the maximum number of characters that the user can enter into the entry field.

The Code Generation option for entry fields will generate code for the setup and query dialog subroutine that will manage the entry field. The Variable entry field should contain the name of a string variable to be used in the parameter list of the dialog box subroutines.

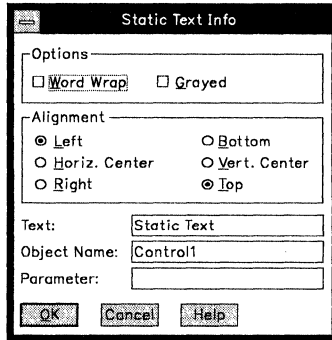
When calling the subroutines, a string variable should be passed for this parameter. The setup subroutine will set the text of the entry field to the contents of this variable. The query subroutine will return the current contents of the entry field in the variable.

The Text entry field contains the initial text to be displayed in the entry field.

When you use the entry field tool, it actually creates two separate controls: an entry field control and a static text control. After you complete the Entry Field Info dialog box, the Static Text Info dialog box appears.

12.4.11 Static Text Info

The Word Wrap check box, in the Static Text Info dialog box shown below, allows you to specify that the text is to be word-wrapped to fit into the control.



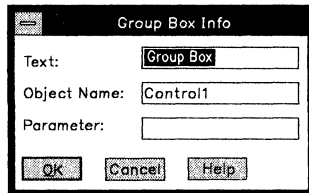
The Grayed check box allows you to specify that the text is to be displayed grayed.

The Alignment options are used to specify how the text is to be aligned within the control.

The Text entry field contains the text to be displayed within the control.

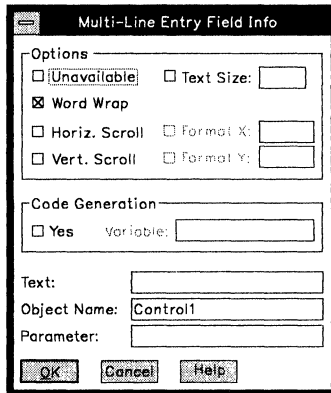
12.4.12 Group Box Info

The Text entry field of the Group Box Info dialog box, shown below, is used to specify the text to be displayed in the top-left corner of the group box.



12.4.13 Multiline Entry Field Info

The Multiline Entry Field Info dialog box, shown below, is used to specify the characteristics of the multiple-line entry field, as well as its code generation options.



The screenshot shows a dialog box titled "Multi-Line Entry Field Info". It contains two main sections: "Options" and "Code Generation".

Options:

- ☐ Unavailable
- ☒ Word Wrap
- ☐ Horiz. Scroll
- ☐ Vert. Scroll
- ☐ Text Size: []
- ☐ Format X: []
- ☐ Format Y: []

Code Generation:

- ☐ Yes Variable: []

Below these sections are three text input fields:

- Text: []
- Object Name: Control1
- Parameter: []

At the bottom are three buttons: OK, Cancel, and Help.

The Unavailable check box is used to specify whether the initial state of the multiple-line entry field is enabled or disabled. When a multiple-line entry field is unavailable, a user cannot enter text into it.

The Text Size check box and entry field specify whether the multiple-line entry field has a text limit, and if so, what it is.

The Word Wrap check box is used to specify whether text will automatically wrap upon entry.

The Scroll options are used to specify whether the multiple-line entry field will have any scroll bars. If a scroll bar is used, the maximum extent of the formatting box can also be specified using the Format check box/entry field combinations. Format sizes are specified in pixels. If no format is specified, the text extent is unlimited.

The Code Generation option for multiple-line entry fields will generate code for the setup and query dialog box subroutines that will manage the entry field. The Variable entry field should contain the name of a string variable to be used in the parameter list of the dialog box subroutines.

When calling the subroutines, a string variable should be passed for this parameter. The setup subroutine will set the text of the multiple-line entry field to the contents of this variable. The query subroutine will return the current contents of the multiple-line entry field in the variable.

The Text entry field contains the initial text to be displayed in the multiple-line entry field.

12.4.14 Combination Box Info

The Combination Box Info dialog box, shown below, allows you to specify the characteristics of the combination box, as well as its code generation options.

Combination Box Info

Options
☒ Simple ☐ Drop Down ☐ Drop Down List

Text Limit: 30 ☒ Unavailable

Sort Order
☒ None ☐ Ascending ☐ Descending

Code Generation
☒ Yes Variable:

Text:

Object Name: Control1

Parameter:

OK Cancel Help

The Options radio buttons allow you to specify the type of combination box: Simple, Drop Down, or Drop Down List.

The Text Limit entry field is used to specify the text limit for the entry field portion of the combination box.

The Unavailable check box is used to specify whether the initial state of the combination box is to be enabled or disabled.

The sort order for the list box portion of the combination box can be specified as None, Ascending, or Descending. If None is specified, list box entries are added according to the order of insertion. If Ascending or Descending is selected, the list box will automatically sort itself independent of the order of insertion.

The Code Generation option for combination boxes will generate code for the setup and query dialog box subroutines that will manager the combination box. The code generated is dependant on the type of combination box.

For simple and drop-down combination boxes, the Variable entry field should contain the name of a string variable to be used in the parameter list of the dialog box subroutines. The setup subroutine will set the text of the combination box entry field to the contents of this variable. The query sub-

routine will return the current contents of the combination box entry field in the variable.

For drop-down lists, the Variable entry field should contain the name of an integer variable to be used in the parameter list of the dialog box subroutines. The setup subroutine will select an item from the list box according to the setting of this variable. The query subroutine will return the currently selected item from the list box in the variable.

The Text entry field contains the initial text to be displayed in the combination box. This text will not be displayed by Layout/CUA in the control.

Chapter 13. Library Subroutines and Functions

13.1 Overview

Libraries are created from files written in an external language (C, Pascal, etc.) and then linked into a dynamic link library.

The libraries supplied with EASEL OS/2 EE contain functions and subroutines. The following libraries are shipped with EASEL OS/2 EE:

Operations	Library	Include File
EASEL OS/2 EE	library "esllib"	esllib.inc
Date	library "datelib"	datelib.inc
File I/O	library "fileio"	fileio.inc
Math	library "mathlib"	mathlib.inc
Message	library "message"	message.inc

13.1.1 Include File

Each of the above libraries has a corresponding include file that you must reference in your EASEL OS/2 EE program before you can invoke any of that library's functions or subroutines.

Use the following syntax to include a file:

```
include FILE_NAME
```

where FILE_NAME is a string literal representing the name of the include file.

The include file contains a declaration line for each function and subroutine comprising a specific library. The declarations are shown under the *Declaration* headings.

13.1.2 Library Subroutines

Library subroutines operate the same way as EASEL OS/2 EE subroutines. Argument list parameters must be specified as variables when the library subroutine is called, since subroutines return values through their parameters. The type and number of arguments after the subroutine name must match the type and number of arguments in the declaration.

To invoke a library subroutine from your EASEL OS/2 EE program, use the **call** statement. The **call** statement syntax is the same for EASEL OS/2 EE and library subroutines:

```
call SUBROUTINE_NAME( [ARG1[, ARG2]... ] )
```

where SUB_NAME is the identifier for a previously defined subroutine.

13.1.3 Library Functions

Library functions differ from subroutines in the following ways:

- They are invoked in the context of an expression or value.
- Their arguments' values cannot be changed in the function.
- They return a value that is used in the evaluation of that expression.

Invoke the library functions by referencing the function name, followed by its arguments, in your EASEL OS/2 EE program. Note the following:

- Arguments can be any arbitrary expression, including constants or variables, but they cannot be functions.
- The number of arguments after the function name must match the number of arguments in the declaration.
- The type of each argument will be converted to the specified type in the declaration.

Use the following syntax to reference a function in an EASEL OS/2 EE program:

```
FUNC_NAME( [ARG1[, ARG2]... ] )
```

where FUNC_NAME is the identifier for the function, and ARGLIST contains expressions separated by commas.

13.2 The EASEL OS/2 EE Library Functions

The EASEL OS/2 EE Library contains the following functions:

Es1QueryEgaColor()

Es1SetEgaColor()

Es1QueryFocus()

Below, *Declaration* shows how the function is declared in the include file (esllib.inc).

Be sure to specify **include esllib.inc** before you specify the function name.

13.2.1 Es1QueryEgaColor() Function

Action: In fullscreen applications, this function returns the EGA or VGA color value for color numbers 18 through 25.

Declaration: function Es1QueryEgaColor(integer:ESL_COLOR_NUMBER)
returns integer
library "esllib"

Invocation: **Es1QueryEgaColor(ESL_COLOR_NUMBER)**

Return Type: **integer**

Remarks: This function is provided for use with fullscreen programs only. It will not work with EASEL OS/2 EE windowed programs.

13.2.2 Es1SetEgaColor() Function

Action: In fullscreen applications, this function sets the EGA or VGA color values for color numbers 18 through 25.

If the display adapter is not an EGA or VGA, the function returns 0; otherwise, it returns a non-zero value.

Declaration: function Es1SetEgaColor(integer:ESL_COLOR_NUMBER,
integer:EGA_VGA_NUMBER)
returns integer
library "esllib"

Invocation: **Es1SetEgaColor(ESL_COLOR_NUMBER, EGA_VGA_NUMBER)**

Return Type: **integer**

Remarks: This function is provided for use with fullscreen programs only. It will not work with EASEL OS/2 EE windowed programs.

13.2.2.1 Default EGA/VGA Color Set

Color Number	EGA/VGA Color Value
18	7 (Light Gray)
19	56 (Dark Gray)
20	11 (Light Blue)
21	5 (Dark Purple)
22	37 (Bright Pink)
23	52 (Orange)
24	38 (Gold)
25	20 (Brown)

13.2.2.2 EGA/VGA Color Values by Color Group

Color Group	EGA/VGA Color Values
Black	0
White	63
Reds	4, 32, 36
Greens	2, 6, 10, 14, 16, 18, 19, 22, 23, 26, 30, 34, 42, 48, 50, 51, 58
Blues	1, 8, 9, 11, 13, 15, 17, 25, 29, 33, 41, 43, 49, 57
Yellows	54, 55, 62
Purples	5, 12, 21, 40, 45, 47, 61
Oranges	38, 52
Cyans	24, 27, 31, 49
Pinks	39, 53, 37
Peaches	44, 46, 60
Grays	7, 56
Blue-grays	3, 35
Brown	20
Tan	28

13.2.3 EslQueryFocus() Function

Action: Returns the name of the EASEL OS/2 EE region that currently has the keyboard input focus.

Declaration: `function EslQueryFocus()
returns string
library "esllib"`

Invocation: `EslQueryFocus()`

Return Type: **string**

Remarks: This function gives the EASEL OS/2 EE program a way to determine where keyboard input should be echoed when there are multiple regions on screen.

Do not call this function in a response to an object, as this may yield unpredictable results.

13.3 The Date Library Functions

In addition to the built-in function **date**, EASEL OS/2 EE provides a Date Library containing the following date functions:

DaySpan()
LocalDate()
LocalTime()
Validdate()
WeekDay()

Specify **include datelib.inc** before you specify the function name. For example:

```
include datelib.inc
```

```
key RetrieveDate at 100 100
  solid green box 100 30
  move 50 15 black text center "Get Date"
  :
response to RetrieveDate
  if ( Validdate(1987,11,24) ) then
    send "Should be Tuesday:" WeekDay(1987,11,24) "\n"
    to errorlog
  end if
```

Declaration, below, shows how the function is declared in the include file (datelib.inc).

13.3.1 DaySpan() Function

Action: Returns a positive integer that is the number of days in the range between two dates. If an invalid date is specified, -1 is returned.

Declaration: function DaySpan(integer:YEAR1, integer:MONTH1,
integer:DAY1, integer:YEAR2,
integer:MONTH2, integer:DAY2)
returns integer
library "datelib"

Invocation: **DaySpan(YEAR1, MONTH1, DAY1, YEAR2, MONTH2, DAY2)**

Where: YEAR1, MONTH1, and DAY1 specify (according to same format as the arguments for **Validdate()**) the earlier date of the range.

YEAR2, MONTH2, and DAY2 specify the later date of the range.

Return Type: **integer**

Example: copy (DaySpan(1989, 6, 5, 1990, 8, 3)) to Num_Of_Days

13.3.2 LocalDate() Function

Action: Returns a string representation of the current date using the preferred local date format. The exact result of this function will depend upon the country in which the program is running and upon user preferences.

Declaration: function LocalTime(integer:TYPE)
returns string
library "datelib"

Invocation: **LocalTime(TYPE)**

Where: TYPE must be either **PMDate** or **OS2Date**. If **PMDate** is specified, the string returned is formatted according to the format specified in the PM Control Panel "Country" settings. (Note that it is possible for the user to change this setting during the execution of a program.) If **OS2Date** is specified, the string returned is formatted according to the COUNTRY setting in the OS/2 CONFIG.SYS file.

Return Type: **string**

Remarks: If an error is encountered, the returned string will be "ERROR".

Example: copy (LocalDate(PMDate)) to Date_String

13.3.3 LocalTime() Function

Action: Returns a string representation of the current time using the preferred local time format. The exact result of this function will depend upon the country in which the program is running and upon user preferences.

Declaration: function LocalTime(integer:TYPE)
returns string
library "datelib"

Invocation: **LocalTime(TYPE)**

Where: TYPE must be either **PMTime** or **OS2Time**. If **PMTime** is specified, the string returned is formatted according to the format specified in the PM Control Panel "Country" settings. (Note that it is possible for the user to change the setting during the execution of a program.) If **OS2Time** is specified, the string returned is formatted according to the COUNTRY setting in the OS/2 CONFIG.SYS file.

Return Type: **string**

Remarks: If an error is encountered, the returned string will be "ERROR".

The OS/2 12-hour time format does not include an AM/PM indicator. If you want an AM/PM indicator in a 12-hour time format, use the **PMTime** format. The AM/PM indicator will be the string specified in the "Country" setting in the PM Control Panel.

Example: copy (LocalTime(PMTime)) to Time_String

13.3.4 Validdatetime() Function

Action: Returns a boolean value that tells whether a given date is valid.

Declaration: function Validdatetime(integer:YEAR, integer:MONTH,
 integer:DAY)
 returns boolean
 library "datelib"

Invocation: **Validdatetime(YEAR, MONTH, DAY)**

Where: YEAR is the full year of the given date, i.e., 1989, 1990, etc.

MONTH is the numerical representation for the month of the given date, i.e., 2 for February, 3 for March, etc.

DAY is the number for the day of the given date.

Return Type: **boolean**

Example: if (Validdatetime(1992, 2, 29)) then
 action GetData
 end if

13.3.5 WeekDay() Function

Action: Returns an uppercase string that is the day of the week corresponding to the specified date. If an invalid date is specified, the NULL string is returned.

Declaration: `function WeekDay(integer:YEAR, integer:MONTH,
integer:DAY)
returns string
library "datelib"`

Invocation: **WeekDay(YEAR, MONTH, DAY)**

Where: YEAR is the full year of the given date, i.e., 1989, 1990, etc.
MONTH is the numerical representation for the month of the given date, i.e., 2 for February, 3 for March, etc.
DAY is the number for the day of the given date.

Return Type: **string**

Example: copy (WeekDay(1989, 3, 8)) to Day_of_Week

13.4 The Math Library Functions and Subroutines

In addition to the basic math operations of addition, subtraction, multiplication, and division built into the language, EASEL OS/2 EE also provides a Math Library containing many useful and frequently required functions. The Math Library consists of 18 math functions and 14 math subroutines.

Math Functions	Math Subroutines
ABS()	MAX()
ARCCOS()	MAXINT()
ARCSIN()	MEAN()
ARCTAN()	MEANINT()
ARCTAN2()	MEDIAN()
CEIL()	MEDIANINT()
COS()	MIN()
E()	MININT()
EXP()	STDDEV()
FLOOR()	STDDEVINT()
LN()	SUM()
LOG()	SUMINT()
MOD()	VARIANCE()
PI()	VARIANCEINT()
POWER()	
SIN()	
SQRT()	
TAN()	

13.4.1 The Math Library Functions

The trigonometric functions are for calculating the sine, cosine, tangent, arcsine, arccosine, and arctangent. **SIN()**, **COS()**, and **TAN()** take a float value representing an angle in degrees and return a float value. **ARCSIN()**, **ARCCOS()**, **ARCTAN()**, and **ARCTAN2()** take a float value and return a float value representing an angle in degrees (to convert to radians, multiply the result by 0.0174532926).

The geometric function, **PI()** takes no arguments and returns the value of Pi. **E()** also takes no arguments and returns the value of e.

The algebraic functions return the value of e (natural log constant), calculate the value of e to a given power, the natural and base-ten logarithms of a given number, the absolute value of a given number, a given number raised to a given power, the square root of a given number, the floor or ceiling of a given number, and the remainder when dividing two given numbers.

The functions all take a float argument, except `e`, which takes no argument. The functions all return a float, except for the floor and ceiling functions, which return integer values.

Example:

```
include mathlib.inc

float SineOfNinety
float Float1 is 2.3

key GetSineOf90 at 100 100
    solid green box 160 30
    move 80 15 black text center "Sine of 90"

action Floor is
    copy FLOOR(Float1) to IResult
    send "FLOOR(" Float1 ") = " IResult ".\n" to
        errorlog

response to start
    action Floor

response to GetSineOf90
    copy SIN(90) to SineOfNinety"
    send "The sine of 90 degrees is: " SineOfNinety "\n"
        to errorlog
```

The Math Library functions are described starting at 13.4.4, “ABS() Function” on page 13-14.

Declaration shows how the function is declared in the include file (mathlib.inc).

Be sure to specify **include mathlib.inc** before you specify the function name.

Note: Math functions cannot be nested.

13.4.2 The Math Library Subroutines

There are seven pairs of statistical subroutines, each with two arguments. The first argument is an array; the second, the result of performing the given operation on the array. One of each pair of subroutines takes an array of integers; the other takes an array of floats. The subroutines find the maximum, minimum, mean, median, standard deviation, variance, and sum of the given array values.

Example:

```
include mathlib.inc

integer IntegerArray[2,3] is -6, 8, 2,
                           3, 4, 1
integer IResult
float FloatArray[3,2] is 2.7, 7.1,
                       4.23, 41.14,
                       0.0, 87.9
float FResult

action MaxInt is
    call MAXINT(IntegerArray, IResult)
    send "Max value of IntegerArray is " IResult "\n" to
        errorlog

action MeanInt is
    call MEANINT(IntegerArray, FResult)
    send "Mean of IntegerArray is " FResult "\n" to
        errorlog

action Sum is
    call SUM(FloatArray, FResult)
    send "SUM of FloatArray is " FResult "\n" to
        errorlog

response to start
    action MaxInt
    action MeanInt
    action Sum
    exit
```

The Math Library subroutines are described starting at 13.4.22, “MAX() Subroutine” on page 13-21.

Declaration shows how the subroutine is declared in the include file (mathlib.inc).

Be sure to specify **include “math.lib.inc”** before you call the subroutine.

Use the **call** action statement to access the math library subroutines.

13.4.3 Error Handling

Error conditions can arise for several of the functions and for the **MEDIAN()**, **MEDIANINT()**, and **SUMINT()** subroutines. In each case, an error message is placed in the errorlog of the form:

Mathlib error detected: ROUTINE_NAME: diagnosis; action.

The diagnosis explains the problem, which for the functions is always an incorrect argument(s). The action explains what the routine did, which for the functions is always to return either zero or negative one, whichever is more likely to stand out. (See the description of each function for the exact error return value.) Thus you can examine at runtime the return value of a function that can encounter an error to determine if an error has occurred, and possibly take remedial action.

The **SUMINT()** subroutine can overflow or underflow, in which case its output parameter is set to zero. The **MEDIAN()** and **MEDIANINT()** subroutines make a copy of the array argument to sort it. The memory for the temporary array is allocated from local storage, and if this allocation generates an error, its output parameter is set to negative one.

13.4.4 ABS() Function

Action: Returns the absolute value of X.

Declaration: function ABS(float:X)
returns float
library "mathlib"

Invocation: **ABS(X)**

Return Type: **float**

13.4.5 ARCCOS() Function

Action: Returns the angle in degrees whose cosine is X.

Declaration: function ARCCOS(float:X)
returns float
library "mathlib"

Invocation: **ARCCOS(X)**

Return Type: **float**

Errors: X must be in the range $-1.0 \leq X \leq 1.0$, or the function returns the value -1 and places the following message in the EASEL OS/2 EE errorlog:

ARCCOS: argument < -1 or > 1; returning -1.0.

13.4.6 ARCSIN() Function

Action: Returns the angle in degrees whose sine is X.

Declaration: function ARCSIN(float:X)
returns float
library "mathlib"

Invocation: **ARCSIN(X)**

Return Type: **float**

Errors: X must be in the range $-1.0 \leq X \leq 1.0$, or the function returns the value -1 and places the following message in the EASEL OS/2 EE errorlog:
ARCSIN: argument < -1 or > 1; returning -10.

13.4.7 ARCTAN() Function

Action: Returns the angle in degrees whose tangent is X.
ARCTAN returns a value in the range -90 to +90 degrees.

Declaration: function ARCTAN(float:X)
returns float
library "mathlib"

Invocation: **ARCTAN(X)**

Return Type: **float**

13.4.8 ARCTAN2() Function

Action: Returns the angle in degrees whose tangent is Y/X.
ARCTAN2 returns a value in the range -180 to +180 degrees.

Declaration: function ARCTAN2(float:Y, float:X)
returns float
library "mathlib"

Invocation: **ARCTAN2(Y, X)**

Return Type: **float**

Errors: X cannot be 0.0, or the function returns the value -1.0 and places the following message in the EASEL OS/2 EE errorlog:

ARCTAN2: second argument = 0; returning -1.0.

13.4.9 CEIL() Function

Action: Returns the smallest integer larger than or equal to X.

Declaration: function CEIL(float:X)
returns integer
library "mathlib"

Invocation: **CEIL(X)**

Return Type: **integer**

Errors: X must be in the range $-2147483648 \leq X \leq 2147483647$, or the function returns the value 0 and places one of the following messages in the EASEL OS/2 EE errorlog:

CEIL: argument > largest integer; returning 0.0.

CEIL: argument < smallest integer; returning 0.0.

13.4.10 COS() Function

Action: Returns the cosine of the argument in degrees.
To convert degrees to radians, multiply degrees by 0.0174532926.

Declaration: function COS(float:DEGREES)
returns float
library "mathlib"

Invocation: **COS(DEGREES)**

Return Type: **float**

13.4.11 E() Function

Action: Returns the constant value of e, which is 2.71828...

Declaration: function E()
returns float
library "mathlib"

Invocation: **E()**

Return Type: **float**

13.4.12 EXP() Function

Action: Returns the exponential function of X (the value which is e to the X power).

Declaration: function EXP(float:X)
returns float
library "mathlib"

Invocation: **EXP(X)**

Return Type: **float**

13.4.13 FLOOR() Function

Action: Returns the largest integer smaller than or equal to X.

Declaration: function FLOOR(float:X)
returns integer
library "mathlib"

Invocation: **FLOOR(X)**

Return Type: **integer**

Errors: X must be in the range -2147483648 <= X <= 2147483647, or the function returns the value 0 and places one of the following messages in the EASEL OS/2 EE errorlog:

FLOOR: argument > largest integer; returning 0.0.

FLOOR: argument < smallest integer; returning 0.0.

13.4.14 LN() Function

Action: Returns the value of the natural logarithm.

Declaration: `function LN(float:X)
returns float
library "mathlib"`

Invocation: **LN(X)**

Return Type: **float**

Errors: X must be a positive, non-zero value, or the function returns the value -1.0 and places the following message in the EASEL OS/2 EE errorlog:
LN: argument <= 0; returning -1.0.

13.4.15 LOG() Function

Action: Returns the value of the logarithm base 10.

Declaration: `function LOG(float:X)
returns float
library "mathlib"`

Invocation: **LOG(X)**

Return Type: **float**

Errors: X must be a positive, non-zero value, or the function returns the value -1.0 and places the following message in the EASEL OS/2 EE errorlog:
LOG: argument <= 0; returning -1.0.

13.4.16 MOD() Function

Action: Returns the remainder produced after dividing X by Y.
For example, MOD(10.0, 3.0) returns 1.0, and MOD(22.0, 2.0) returns 0.0.

Declaration: `function MOD(float:X, float:Y)
returns float
library "mathlib"`

Invocation: **MOD(X, Y)**

Return Type: **float**

Errors: Y cannot equal 0.0, or the function returns the value -1 and places the following message in the EASEL OS/2 EE errorlog:

MOD: second argument = 0; returning -1.0.

13.4.17 PI() Function

Action: Returns the constant value of Pi, which is 3.14159...

Declaration: function PI()
returns float
library "mathlib"

Invocation: **PI()**

Return Type: **float**

13.4.18 POWER() Function

Action: Returns X raised to the Y power.

If Y is 0.0, the function returns 1.0.

If X is negative and Y is not a whole number (that is, Y has a non-zero fractional part), the function computes the value using only the integral part of Y.

Declaration: function POWER(float:X, float:Y)
returns float
library "mathlib"

Invocation: **POWER(X, Y)**

Return Type: **float**

Errors: X cannot be 0.0 and Y cannot be negative, or the function returns the value 0.0 and places the following message in the EASEL OS/2 EE errorlog:

POWER: base = 0 and exponent < 0; returning 0.0.

13.4.19 SIN() Function

Action: Returns the sine of the argument in degrees. To convert degrees to radians, multiply degrees by 0.0174532926.

Declaration: function SIN(float:DEGREES)
returns float
library "mathlib"

Invocation: **SIN(DEGREES)**

Return Type: **float**

13.4.20 SQRT() Function

Action: Returns the square root of X.

Declaration: function SQRT(float:X)
returns float
library "mathlib"

Invocation: **SQRT(X)**

Return Type: **float**

Errors: X cannot be negative, or the function returns the value 0 and places the following message in the EASEL OS/2 EE errorlog:

SQRT: argument < 0; returning 0.0.

13.4.21 TAN() Function

Action: Returns the tangent of the argument in degrees. To convert degrees to radians, multiply degrees by 0.0174532926.

Declaration: function TAN(float:DEGREES)
returns float
library "mathlib"

Invocation: **TAN(DEGREES)**

Return Type: **float**

Errors: The argument must not be 90 or 270 degrees, or the function returns the value 0 and places the following message in the EASEL OS/2 EE errorlog:

TAN: not defined for 90 or 270 degrees; returning 0.0.

13.4.22 MAX() Subroutine

Action: Returns the largest value in the float array. The value is returned in the MAXIMUM argument.

Declaration: subroutine MAX(float:ARRAY_IN, float:MAXIMUM)
library "mathlib"

Invocation: **MAX(ARRAY_IN, MAXIMUM)**

13.4.23 MAXINT() Subroutine

Action: Returns the largest value in the integer array. The value is returned in the MAXIMUM argument.

Declaration: subroutine MAXINT(integer:ARRAY_IN, integer:MAXIMUM)
library "mathlib"

Invocation: **MAXINT(ARRAY_IN, MAXIMUM)**

13.4.24 MEAN() Subroutine

Action: Returns the mean (or average) of all the values in the float array. The value is returned in the MEAN_VALUE argument.

Declaration: subroutine MEAN(float:ARRAY_IN, float:MEAN_VALUE)
library "mathlib"

Invocation: **MAXINT(ARRAY_IN, MEAN_VALUE)**

13.4.25 MEANINT() Subroutine

Action: Returns the mean (or average) of all the values in the integer array. The value is returned in the MEAN_VALUE argument.

Declaration: subroutine MEANINT(integer:ARRAY_IN, float:MEAN_VALUE)
library "mathlib"

Invocation: **MEANINT(ARRAY_IN, MEAN_VALUE)**

13.4.26 MEDIAN() Subroutine

Action: Returns the median of all the values in the float array. The value is returned in the MEDIAN_VALUE argument.

The median is defined as follows:

For an odd number of elements in an array, it is the middle value when all elements are sorted by value.

For an even number of elements in the array, the median is the average of the two middle values after all elements in the array are sorted by value.

Declaration: subroutine MEDIAN(float:ARRAY_IN, float:MEDIAN_VALUE)
library "mathlib"

Invocation: **MEDIAN(ARRAY_IN, MEDIAN_VALUE)**

Errors: If there is insufficient memory to allocate an array buffer, the subroutine sets MEDIAN_VALUE to -1 and places the following message in the EASEL OS/2 EE errorlog:

MEDIAN: memory allocation error; setting value to -1.

13.4.27 MEDIANINT() Subroutine

Action: Returns the median of all the values in the integer array. The value is returned in the MEDIAN_VALUE argument.

The median is defined as follows:

For an odd number of elements in an array, it is the middle value when all elements are sorted by value.

For an even number of elements in the array, the median is the average of the two middle values after all elements in the array are sorted by value.

Declaration: subroutine MEDIANINT(integer:ARRAY_IN, float:MEDIAN_VALUE)
library "mathlib"

Invocation: **MEDIANINT(ARRAY_IN, MEDIAN_VALUE)**

Errors: If there is insufficient memory to allocate an array buffer, the subroutine sets MEDIAN_VALUE to -1 and places the following message in the EASEL OS/2 EE errorlog:

MEDIANINT: memory allocation error; setting value to -1.

13.4.28 MIN() Subroutine

Action: Returns the smallest value in the float array. The value is returned in the MINIMUM argument.

Declaration: subroutine MIN(float:ARRAY_IN, float:MINIMUM)
library "mathlib"

Invocation: **MIN**(ARRAY_IN, MINIMUM)

13.4.29 MININT() Subroutine

Action: Returns the smallest value in the integer array. The value is returned in the MINIMUM argument.

Declaration: subroutine MININT(integer:ARRAY_IN, integer:MINIMUM)
library "mathlib"

Invocation: **MININT**(ARRAY_IN, MINIMUM)

13.4.30 STDDEV() Subroutine

Action: Returns the standard deviation of all values in the float array. The value is returned in the STANDARD_DEV argument.

Declaration: subroutine STDDEV(float:ARRAY_IN, float:STANDARD_DEV)
library "mathlib"

Invocation: **STDDEV**(ARRAY_IN, STANDARD_DEV)

13.4.31 STDDEVINT() Subroutine

Action: Returns the standard deviation of all values in the integer array. The value is returned in the STANDARD_DEV argument.

Declaration: subroutine STDDEVINT(integer:ARRAY_IN, float:STANDARD_DEV)
library "mathlib"

Invocation: **STDDEVINT**(ARRAY_IN, STANDARD_DEV)

13.4.32 SUM() Subroutine

Action: Returns the sum of all values in the float array. The value is returned in the SUMMATION argument.

Declaration: subroutine SUM(float:ARRAY_IN, float:SUMMATION)
library "mathlib"

Invocation: **SUM(ARRAY_IN, SUMMATION)**

13.4.33 SUMINT() Subroutine

Action: Returns the sum of all values in the integer array. The value is returned in the SUMMATION argument.

Declaration: subroutine SUMINT(integer:ARRAY_IN, integer:SUMMATION)
library "mathlib"

Invocation: **SUMINT(ARRAY_IN, SUMMATION)**

Errors: If the result exceeds the largest integer (2147483647) or the smallest integer (-2147483648), the subroutine sets SUMMATION to 0 and places one of the following messages in the EASEL OS/2 EE errorlog:

result > largest integer; setting value to 0.

result < smallest integer; setting value to 0.

13.4.34 VARIANCE() Subroutine

Action: Returns the variance of all values in the float array. The value is returned in the VARIANCE argument.

The equation used is based upon the number of elements in the array, n, and not (n-1).

Declaration: subroutine VARIANCE(float:ARRAY_IN, float:VARIANCE)
library "mathlib"

Invocation: **VARIANCE(ARRAY_IN, VARIANCE)**

13.4.35 VARIANCEINT() Subroutine

Action: Returns the variance of all values in the integer array. The value is returned in the VARIANCE argument.

The equation used is based upon the number of elements in the array, n, and not (n-1).

Declaration: subroutine VARIANCEINT(integer:ARRAY_IN,
float:VARIANCE)
library "mathlib"

Invocation: **VARIANCEINT(ARRAY_IN, VARIANCE)**

13.5 The File I/O Library

13.5.1 Overview

This section discusses the File I/O Library in the following order:

1. Introduction to the File I/O Library
2. Record Definition
3. Open and Close Subroutines
4. Input Subroutines
5. Output Subroutines
6. Parameter Adjustment Subroutines
7. Error Handling Subroutine

13.5.2 The File I/O Library Subroutines

The File I/O Library consists of several subroutines that facilitate reading and writing ASCII files from your EASEL OS/2 EE program. Some of the tasks you can perform with the File I/O subroutines are:

- Create a file.
- Append to a file.
- Read lines from a file sequentially.
- Access any specified line in a file.
- Read a single line into a single string variable.
- Read a single line into a combination of string, integer, and float variables according to a specification called a record.
- Write a single string to a file.
- Write a record containing a combination of string, integer, and float variables.
- Control parameters that enable tradeoffs between program speed and size.

Files are processed as follows:

1. The file is *opened* with specific parameters that identify the file (the file identifier and the file name), and that specify whether the file is to be read, written, or appended (the access mode).
2. The file is then processed by calling one or more of the *Input* or *Output subroutines*, depending on whether you are reading or writing to the file.
3. After processing is complete, the file is *closed*.

The Input and Output subroutines treat ASCII files as if they are composed of lines. They enable you to read or write lines of the file sequentially or randomly, or as a combination of both methods. Files consisting of fields of fixed size in fixed locations can be mapped into separate variables by using

an appropriate **record** definition. Records can also be written, so that files can be created in a specific fixed format.

The fields of the record are defined in the **record** definition statement, which must be declared before file processing begins. The record name is then used as a parameter for certain Input and Output subroutines that read or write records.

The parameter types that are passed to any of the File I/O subroutines are specific to that subroutine, which is a special parameter type. It is used as the unique identifier for files in all I/O subroutines. Parameters can be either input or output. Input parameters already have values when the subroutine is called, whereas output parameters have a value assigned during the execution of the subroutine.

If a File I/O Library subroutine encounters an error, **errorlevel** is set to the number of the error found. **errorlevel** contains a value of zero when no errors are detected. The **GetError** subroutine sets a string variable to the error message associated with the error level number passed to it.

13.5.3 The Record Definition

Records are used in conjunction with the File I/O Library subroutines to break up individual input lines into many fields and to format data. On input, a record maps a line of data into specific variables representing data fields. On output, a record determines output lines from many different variables.

All the variable fields in a **record** definition must be specified previously, except for a special field called **filler**. On input, **filler** indicates portions of the input line that should be skipped over. On output, **filler** indicates the placement of “white space” in the output.

Records themselves can contain records, but a record cannot contain itself, either directly or indirectly. Records can also contain array references, but not simply an array name. Subscripts can be variables.

Records are defined as follows:

Model

```
record RECORD_NAME is
  [filler FILL_WIDTH]
  { VARIABLE_NAME VAR_WIDTH[, PREC] |
    VARIABLE_NAME[ N1[, N2] ] VAR_WIDTH[, PREC] |
    record PD_RECORD_NAME }
  [ filler FILL_WIDTH |
    VARIABLE_NAME VAR_WIDTH[, PREC] |
    VARIABLE_NAME[ N1[, N2] ] VAR_WIDTH[, PREC] |
    record PD_RECORD_NAME ]...
end record
```

13.5.3.1 Filler

Specify a filler using:

```
filler WIDTH
```

filler is a keyword. The WIDTH specifier is a positive integer constant (greater than zero) denoting the number of columns skipped before the next field begins. If you specify **filler** as the last field in a record, it is ignored. Note that **filler** cannot be the only specification contained in a record.

13.5.3.2 Variable Name

Specify a variable name using:

```
VARIABLE_NAME WIDTH[, PREC]
```

VARIABLE_NAME is either a string, integer, or float variable, followed by its WIDTH specifier, and optionally, a PREC specifier.

WIDTH is a positive or negative integer constant specifying the number of columns in the field (including the decimal point and the PREC for floats). On input, the sign of the WIDTH specifier is ignored. On output, the sign of the WIDTH specifier indicates the justification of the value in the output field:

Sign	Alignment in Output Field	Blanks appear
positive	right	to left of value
negative	left	to right of value

PREC (PRECISION) is a positive integer constant specifying the number of digits to the right of the decimal point for a float variable. If no PRECISION

is specified for a float variable in the **record** definition, zero is used. (PRECISION will be ignored if specified for either string or integer variables.)

Note: Precision in a record is different from EASEL OS/2 EE precision. EASEL OS/2 EE defaults to a precision of two without a specified precision for floating point. Thus, actions performed on float variables read in by the Input subroutines will contain a decimal point and two zeros. If the floating point variable is read in with a precision greater than two, but the EASEL OS/2 EE program defaults to a precision of two, the digits after the first two will be truncated.

13.5.3.3 Array Variable Reference

Specify an array variable reference using:

VARIABLE_NAME[N1[, N2]...] WIDTH[, PREC]

VARIABLE_NAME[N1[, N2]...] is an array variable reference. The array must be a string, integer, or float array, and the reference must be specific (that is, to a single element in the array). The array can be multidimensional. Subscripts can be either positive integer constants or integer variables. (See 13.5.3.5, "Example 1" on page 13-30.)

13.5.3.4 Record

Specify a record using:

record RECORD_NAME

record is a keyword. RECORD_NAME is the name of another record that has been previously defined.

A record cannot contain itself, either directly or indirectly. The limit on nesting depends on available memory, but is essentially unlimited. There is no WIDTH specifier associated with a record specification, since the fields are already defined in the definition of the record being referenced. (See 13.5.3.6, "Example 2" on page 13-31.)

13.5.3.5 Example 1

A useful feature of the **record** definition is that the record itself can contain a variable specification that can be used as a subscript later in the same record. In the following example, LineNum is specified as an integer constant, and is later used as the subscript for Lines:

```
fileid FileID
string FileName is "input.txt"
string Read is "read-only"
integer LineNum
integer Counter is 1
string Lines[100]

record InputLine is
  LineNum      2
  filler       1
  Lines[LineNum] 40
end record
```

The file "input.txt" is shown below:

```
02 This transaction applies to 2.
05 This transaction applies to 5.
08 This transaction applies to 8.
03 This transaction applies to 3.
01 This transaction applies to 1.
08 This transaction applies to 8.
04 This transaction applies to 4.
10 This transaction applies to 10.
11 This transaction applies to 11.
:
```

The following example is EASEL OS/2 EE code that reads the first 10 lines of the input file:

```
response to start
  call Open(FileID,FileName,Read)
  while (Counter < 11)
    loop                                # read first 10 lines
      call ReadNextRecord(FileID,InputLine)
      copy (Counter + 1) to Counter
    end loop
  call Close(FileID)
```

The result is that:

Lines[1] contains the string beginning on line 5, starting at column 4, up to 40 characters ("This transaction applies to 1.")

Lines[2] contains the string beginning on line 1 starting at column 4, up to 40 characters ("This transaction applies to 2.") and so on.

13.5.3.6 Example 2

The following example shows how a **record** definition references another **record** definition:

```
integer Salary
string FirstName MiddleName LastName
record Name is
    FirstName    10
    MiddleName   10
    LastName     10
end record
string Street Apartment City State Zip Extension
record Address is
    Street       20
    filler        1
    Apartment     5
    City          15
    filler        1
    State         15
    Zip           5
    filler        1
    Extension     4
end record
record EmployeeRec is
    record Name
    record Address
    filler 5
    Salary 10
end record
```

The last record, EmployeeRec, references the Name record and the Address record, which were both defined previously.

13.5.4 The Open() and Close() Subroutines

The **OpenFile()** and **CloseFile()** subroutines perform the necessary overhead required when opening and closing a file, and are defined in the include file (**fileio.inc**) according to the following syntax.

```

Declarations:      subroutine OpenFile(fileid:FILE_ID, string:FILE_NAME,
                        string:ACCESS_MODE)
                        library "fileio"

                        subroutine CloseFile(fileid:FILE_ID)
                        library "fileio"

```

Where: FILE_ID must be declared previously by the **fileid** statement:

```
fileid FILE_ID
```

FILE_ID is a user-defined name that follows the rules for an EASEL OS/2 EE *identifier*. (An example of **fileid** declaration is included in the file listing in 13.5.4.1, "Call" on page 13-33.)

The file identifier is actually a *file descriptor*, or handle, which is used internally to identify the file. This provides a more efficient means of accessing the file. A call to **OpenFile()** associates the file identifier with the file to be opened. Therefore, whenever I/O is to be performed on the file, the file identifier is used instead of the file name, and it is always the first argument in all of the File I/O Library subroutines. A call to **CloseFile()** disassociates the fileid from the file.

FILE_NAME is a string variable that is the name of the file to be opened. If a file is opened twice for output, the one opened last will be the one that is actually written. Using **OpenFile()** twice for reading the same file is acceptable, as long as each call to **OpenFile** specifies a unique file identifier. Opening a file for both reading and writing will produce different results depending on which one is opened and closed first. (See 13.5.4.2, "Example" on page 13-33.)

ACCESS_MODE is a string variable that contains one of three file modes, indicating how the file can be accessed after it is opened:

"read-only", "read", or "r"

A file can be opened read-only, indicating the file will not be modified. An error message is generated when an attempt is made to open a nonexistent file with read access specified.

"write-only", "write", or "w"

A file can be opened write-only, indicating the file will be created if it does not exist. If an existing file is opened write-only, and it is not previously set as write-protected, the contents of the file will be erased. An error message is generated when a write-protected file is opened with write access specified.

"append" or "a"

A file can be opened with append access, indicating the file will be preserved, and any output produced is appended to the end of the file. The file is created if it does not exist. An error message is generated when a write-protected file is opened with append access specified.

13.5.4.1 Call

A call to **OpenFile()** or **CloseFile()** is made according to the following syntax:

Model

```
call { OpenFile( FILE_ID, FILE_NAME, ACCESS_MODE ) |  
      CloseFile( FILE_ID ) }
```

13.5.4.2 Example

The following example of an EASEL OS/2 EE program shows the declaration of the FILE_ID, FILE_NAME, and ACCESS_MODE variables, and how they are used in calls to the **OpenFile()** and **CloseFile()** subroutines:

```
fileid  FileID  
fileid  FileIDout  
string  FileName is "input.txt"  
string  FileNameOut is "output.txt"  
string  ReadIn is "read-only"  
string  WriteTo is "write-only"  
include "fileio.inc"  
key File 50 50  
:  
response to File is  
  call OpenFile (FileID, FileName, ReadIn)  
  if (errorlevel>0) then  
                                     # appropriate error  
                                     # handling  
  
:  
:
```

```

        end if
    call OpenFile (FileIDout, FileNameOut, WriteTo)    # using string
    if (errorlevel>0) then                            # constants
                                                    # appropriate error
                                                    # handling

:
    end if

                                                    # perform read and
                                                    # write operations

:
call CloseFile (FileID)
    if (errorlevel>0) then

                                                    # appropriate error
                                                    # handling

:
    end if
call CloseFile (FileIDout)
    if (errorlevel>0) then

                                                    # appropriate error
                                                    # handling

:
    end if

```

13.5.4.3 OpenFile() Subroutine Error Messages

The following errors can be returned by the **OpenFile()** subroutine.

errorlevel	Error Message/Explanation
------------	---------------------------

- | | |
|---|---|
| 1 | Error locking global memory.
Indicates a serious problem. Contact your support representative. |
| 3 | Unrecognized file mode specified.
Make sure file access mode begins with either "r", "w", or "a". |
| 4 | Attempt to open a file that does not exist. |
| 5 | Attempt to open write-protected file with mode "w".
Check file's access permission. |
| 6 | Attempt to open write-protected file with mode "a".
Check file's access permission. |
| 7 | Insufficient memory to satisfy allocation request.
System memory is too low to allocate. |

- 8 File handle is invalid.**
Unlikely error. Contact your support representative.
- 24 File has already been opened.**
Occurs when you call **OpenFile()** twice using the same file identifier. Either use a unique file identifier for each call to **OpenFile()**, or call **CloseFile()** for that file identifier before you open it again.

13.5.4.4 CloseFile() Subroutine Error Messages

The following errors can be returned by the **CloseFile()** subroutine.

errorlevel	Error Message/Explanation
1	Error locking global memory. Indicates a serious problem. Contact your support representative.
2	File has not been opened. Occurs when you try to close a file that has not been opened.
8	File handle is invalid. Unlikely error. Contact your support representative.

13.5.5 Input Subroutines

The Input subroutines allow you to read the contents of an ASCII file. The four Input subroutines are **ReadNext()**, **ReadNextRecord()**, **ReadLineNumber()**, and **ReadRecordAtLine()**. Their use depends on whether the lines of the file will be read sequentially or randomly, and where the lines will be stored—in a single variable or in a record consisting of several variables. Only one line at a time is read.

Input Subroutine	Input Method	Output Format
ReadNext()	Sequential	Variable
ReadNextRecord()	Sequential	Record
ReadLineNumber()	Random	Variable
ReadRecordAtLine()	Random	Record

Each subroutine is defined in the include file (**fileio.inc**) according to the following syntax.

```

Declarations:      subroutine ReadNext(fileid:FILE_ID, string:TARGET)
                      library "fileio"

                      subroutine ReadNextRecord(fileid:FILE_ID, record:RECORD_NAME,
                      library "fileio"

                      subroutine ReadLineNumber(fileid:FILE_ID,
                      integer:LINE_NUMBER,
                      string:TARGET)
                      library "fileio"

                      subroutine ReadRecordAtLine(fileid:FILE_ID,
                      integer:LINE_NUMBER,
                      record:RECORD_NAME )
                      library "fileio"

```

Where: FILE_ID must be previously declared with the **fileid** statement, and it must be the same as the file identifier used when the file was opened. (See 13.5.4, "The Open() and Close() Subroutines" on page 13-31.)

TARGET in **ReadNext()** and **ReadLineNumber()** is a string variable, which receives the data from the appropriate input line of the file. The target string will not contain any newline characters (\n) or carriage returns (\r).

The LINE_NUMBER is an integer variable that contains the number of the line to be read. This parameter must be an integer variable greater than zero.

RECORD_NAME in **ReadNextRecord()** and **ReadRecordAtLine()** is the name of a record. RECORD_NAME must be previously defined by the **record** definition statement (see 13.5.7, "Parameter Adjustment Subroutines" on page 13-44). The variables specified in the **record** definition receive the data from the appropriate input line of the file.

Remarks: All Input subroutines keep track of where the next line of the file begins, so **ReadNext()** and **ReadNextRecord()** can be called at any time, provided there are still lines in the file to be read. **ReadLineNumber()** and **ReadRecordAtLine()** find the requested line, if it is in the file, according to its position from the beginning of the file. For example, if you specify a LINE_NUMBER of 3, the third line in the file will be read, etc.

13.5.5.1 Buffers

All Input subroutines use an internal buffer to read from files. The sequential subroutines (**ReadNext()** and **ReadNextRecord()**) fill a buffer starting at the beginning of the next line, and expect to find the end of the line in the buffer. If they do not, they keep increasing the buffer's size until they either find the end of the line or they run out of system memory. The random access subroutines (**ReadLineNumber()** and **ReadRecordAtLine()**) first scan the file to find the beginning of the specified line by repeatedly filling and scanning the file's buffer until the specified line is found. A buffer size of 1K has been found to be optimal, and increments of 1K are recommended when increasing or decreasing the buffer's size (see 13.5.7, "Parameter Adjustment Subroutines" on page 13-44).

The Input subroutines count lines as they scan the file, and keep track of where each scanned line is located. This information, called a file's index, is stored in a buffer in memory. With a large file, the index could grow unreasonably large. Therefore, each file has a limit on its index size. When the limit is reached, **ReadLineNumber()** and **ReadRecordAtLine()** reduce the index size by storing only half as many line locations while scanning the file. The index then stores the beginning position of every other line. This process of reducing the index size is repeated as necessary to keep each file's index under its specified limit, which starts at 1024 bytes (1K) total space, or 256 line beginning positions, since each index takes up 4 bytes. You can alter the limit by calling **SetIndexSize()** for a specified file.

13.5.5.2 Line Terminator

The Input subroutines interpret the end of a line in an ASCII file according to the code used by the application to terminate the line when the file was created. Well-behaved applications and File I/O Output subroutines create files with a carriage return/newline (`\r\n`) sequence to indicate the end of a line. In this case, the File I/O Input subroutines terminate the line immediately after the newline character (`\n`). An example follows:

This will be on line 1\r\nThis will be on line 2

is interpreted as:

This will be on line 1\r\n

This will be on line 2

Conversely, a line termination sequence of `\n\r` causes the line to be broken immediately after the carriage return (`\r`). Because not all files are created using a line termination sequence of carriage return/newline, the File I/O Library subroutines can interpret lines that are terminated with a

carriage return (\r) or a newline (\n) used individually. For example:

```
This is line 1\r\r\rThis is line 4
```

is interpreted as:

```
This is line 1\r
\r
\r
This is line 4
```

Also notice in the above example that if a file ends without a final line terminator, the last line of the file is still included in the line number count. When lines are terminated with a newline character (\n), they are interpreted the same as in the above example.

When an “end of file” terminator (^Z, \032 octal, or 3A hex) is encountered, the file is terminated immediately, and any lines following the end of file terminator are not included in the line number count.

If there is no data to fill a field, it is assigned a reasonable default value instead. For integers and floats, the default value is 0, and for strings, the default value is the null string (""). Strings are read into a variable exactly as they appear in the input line. Integers are read according to the following model:

[whitespace] [sign] digits

Therefore, after digits are encountered, the next non-digit terminates the value. For example:

```
fileid FileID
string FileName is "input.txt"
string Mode is "r"
integer X Y
record In is
    X 10
    Y 10
end record
:
call OpenFile( FileID, FileName, Mode )
call ReadNextRecord( FileID, In )
```

The input file “input.txt” is shown below:

```
1 2 3 4 5 6 7 8 9 10
```

The result is that the variable X is assigned the value 1, and the variable Y is assigned the value 6. No error is generated since this is reasonable behavior. Floats are read according to the following model:

[whitespace] [sign] [digits] [.digits]
[{e|E} [sign] [digits]]

13.5.5.3 Call

A call to **ReadNext()**, **ReadNextRecord()**, **ReadLineNumber()**, or **ReadRecordAtLine()** is made according to the following syntax:

Model

```
call { ReadNext( FILE_ID, TARGET ) |  
      ReadNextRecord( FILE_ID, RECORD_NAME ) |  
      ReadLineNumber( FILE_ID, LINE_NUMBER, TARGET ) |  
      ReadRecordAtLine( FILE_ID, LINE_NUMBER, RECORD_NAME ) }
```

13.5.5.4 Example

The following example of an EASEL OS/2 EE program shows the declaration of FILE_ID:

```
fileid   FileID  
fileid   FileIDout  
string   FileName is "input.txt"  
string   FileNameOut is "output.txt"  
string   Read is "read-only"  
string   WriteTo is "Write"  
string   Target  
integer  LineNum  
string   NewLine is "\n"  
record   LineOut is  
    LineNum    5  
    Target     20  
    NewLine    1  
end record
```

```
include "fileio.inc"
```

```
# user types in line number desired, and the first 20 characters of that  
# line and the one after it are read in and written out.  
# To be brief, error handling is omitted.
```

```
response to line from keyboard is  
    call OpenFile( FileID, FileName, Read )  
    call OpenFile( FileIDout, FileNameOut, WriteTo )  
    copy input to LineNum  
    call ReadLineNumber( FileID, LineNum, Target )  
    call WriteRecord( FileIDout, LineOut )  
    call ReadNext( FileID, Target )  
    copy(1 + LineNum) to LineNum  
    call WriteRecord( FileIDout, LineOut )  
    call CloseFile( FileID )  
    call CloseFile( FileIDout )
```

13.5.5.5 Input Subroutines Error Messages

The following errors can be returned by any of the Input subroutines.

errorlevel	Error Message/Explanation
1	Error locking global memory. Indicates a serious problem. Contact your support representative.
2	File has not been opened. An attempt was made to read the file before it was opened.
8	File handle is invalid. Unlikely error. Contact your support representative.
9	Attempt to read at end-of-file. All lines have been read.
11	Bad offset on seek. Unlikely error. Contact your support representative.
12	Line number specified is not in the file.
13	Line too big to fit in maximum buffer size. Although an attempt was made to increase the buffer's size to fit in a complete line, system memory ran out before the current line could be accommodated. Make sure lines are terminated by a carriage return (\r) or newline (\n), otherwise the entire file will be interpreted as one line.
18	Attempt to read from file opened for writing.
19	File handle invalid or file is locked. The file could have been locked by some other application. Invalid handle is extremely unlikely.
20	Cannot increase buffer size. Although an attempt was made to increase the buffer's size to fit in a complete line, system memory ran out before the current line could be accommodated. Make sure lines are terminated by a carriage return (\r) or newline (\n), otherwise the entire file will be interpreted as one line.

In addition to the above errors, **ReadNextRecord()** and **ReadRecordAtLine()** can also return the following errors:

errorlevel	Error Message/Explanation
21	Array subscript reference out of range. One of the array subscripts in the record is greater than the highest allowable subscript for that dimension.

- 22 **Inappropriate field in record (must be integer, float, or string).**
Unlikely error (usually caught at compile time). Contact your support representative.
- 23 **Array field in record has too few subscripts.**
The array referenced in the record has more dimensions than the number of subscripts specified.

13.5.6 Output Subroutines

The Output subroutines allow you to create an ASCII file on disk. The **WriteString()** subroutine outputs a string value to a file. The **WriteRecord()** subroutine outputs the variables specified by the **record** definition to a file.

Declarations: subroutine WriteRecord(fileid:FILE_ID, record:RECORD_NAME)
 library "fileio"

 subroutine WriteString(fileid:FILE_ID, string:SOURCE)
 library "fileio"

Where: FILE_ID is a string variable for the fileid.

RECORD_NAME in **WriteRecord()** is the name of the source record.

SOURCE in **WriteString()** is a string variable that can contain newline characters (\n) and carriage returns (\r).

Remarks: The SOURCE string, or strings in the source record, can have embedded newline characters, and the record can have strings that have embedded newlines as fields, so that a single call to an Output subroutine can result in several lines being written to the file.

13.5.6.1 Buffers

The **WriteRecord()** subroutine assembles the output line(s) in its buffer and then, for the sake of efficiency, outputs them to disk at one time. Therefore, adequate buffer size is essential. A 1K buffer is the default, but the buffer size can be adjusted (see 13.5.7, "Parameter Adjustment Subroutines" on page 13-44). If the buffer size is not large enough for all the fields of a record to fit evenly, the buffer is repeatedly increased to the next larger 1K increment until they fit.

13.5.6.2 Call

A call to **WriteRecord()** or **WriteString()** is made according to the following syntax:

Model

```
call { WriteRecord( FILE_ID, RECORD_NAME ) |  
      WriteString( FILE_ID, SOURCE ) }
```

13.5.6.3 Example

The following example of an EASEL OS/2 EE program shows the declaration of FILE_ID, SOURCE, and RECORD_NAME, and how they are used in calls to the **WriteString()** and **WriteRecord()** subroutines.

```
fileid    FileID  
string    FileNameOut is "myfile.txt"  
string    FileMode  
string    Source is "This is a string."  
string    Line is "This is a line."  
string    NewLine is "\n"  
record    OutRec is  
    filler    10  
    Line      80  
    NewLine    1  
end record  
  
include "fileio.inc"  
  
# To be brief, error handling is omitted  
  
response to line "Open Myfile" from keyboard  
    copy "w" to FileMode  
    call OpenFile( FileID, FileNameOut, FileMode )  
  
response to line "Close Myfile" from keyboard  
    call CloseFile( FileID )  
  
response to line "Write Record" from keyboard  
    call WriteRecord( FileID, OutRec )  
  
response to line "Write String" from keyboard  
    call WriteString( FileID, Source )
```

13.5.6.4 Output Subroutines Error Messages

The following errors can be returned by either of the output subroutines.

errorlevel	Error Message/Explanation
1	Error locking global memory. Serious problem. Contact your support representative.
2	File has not been opened. An attempt was made to write to a file before it was opened.
8	File handle is invalid. Unlikely error. Contact your support representative.
14	Attempt to write to file opened for reading only.
15	Insufficient space on disk to satisfy write request. The disk where the file is being written is full.
16	Unknown error code returned from C runtime library routine. Unlikely error. Contact your support representative.
17	String too long to write out (greater than 65534). Unlikely error. Contact your support representative.

In addition to the above errors, **WriteRecord()** can also return the following errors.

errorlevel	Error Message/Explanation
13	Line too big to fit in maximum buffer size.
20	Cannot increase buffer size.
21	Array subscript reference out of range. An array subscript in the record is greater than the highest allowable subscript for that dimension.
22	Inappropriate field in record (must be integer, float, or string). Unlikely error (usually caught at compile time). Contact your support representative.
23	Array field in record has too few subscripts. The array referenced in the record has more dimensions than the number of subscripts specified.

13.5.7 Parameter Adjustment Subroutines

SetBufferSize() and **SetIndexSize()** are subroutines that allow you to modify the buffers used by the File I/O Input subroutines to keep track of the number of lines in a file. These two modifications allow you to adjust the sizes of the buffers to optimize the speed at which the file is read, while keeping the buffer sizes as small as possible.

Program speed can be improved in some cases by increasing the buffer size or the index buffer size for one or more files. Conversely, the amount of memory needed can be reduced by decreasing the buffer size or the index buffer size for one or more files. The default values for both parameters are 1K. If you choose to resize the buffers, make the modifications only once, just after opening the file.

The **SetTabSize()** subroutine allows you to control the number of columns between tab stops. The default is 8. The tabs are especially important in records as the layout of the line is specific, and how any tabs present are translated into blanks can be crucial. Although tab sizes of up to 64K are supported, such large tab sizes are not recommended.

The **SetBufferSize()**, **SetIndexSize()**, and **SetTabSize()** subroutines are defined in the include file (**fileio.inc**) according to the following syntax.

```
Declarations:    subroutine SetBufferSize(fileid:FILE_ID,
                      integer:BUFFER_SIZE)
                      library "fileio"

                      subroutine SetIndexSize(fileid:FILE_ID,
                      integer:INDEX_SIZE)
                      library "fileio"

                      subroutine SetTabSize(fileid:FILE_ID,
                      integer:TAB_SIZE)
                      library "fileio"
```

Where: **FILE_ID** is a string variable for the file identifier.

BUFFER_SIZE is an integer variable containing the buffer size in bytes.

INDEX_SIZE is an integer variable containing the index buffer size in bytes.

TAB_SIZE is the number of columns between tab stops (greater than 0 and less than 64K).

Remarks: Decreasing the buffer size can cause a degradation of response time. Since the buffer fills up more often, the Input subroutines must scan the file multiple times. If the buffer is sized to less than the length of a line, the Input subroutines will attempt to increase the buffer to the next larger increment of 1K. If unsuccessful, the line will not be read.

Buffer size also affects **WriteRecord()**, which first collects all of the fields of the record in the buffer, and then writes the buffer's contents all at once. If the buffer is too small to hold the whole record, **WriteRecord()** will attempt to increase the buffer to the next larger increment of 1K. If unsuccessful, the record will not be written.

Increasing the buffer size will improve speed, but always increase the buffer size to a multiple of 1K, or response time may actually be slowed. The buffer size can be a maximum of 64K. If a request is made for more than 64K, a buffer of 64K will be allocated.

Decreasing the index size can be beneficial if you are using the sequential Input subroutines, since they do not use the index. Once the index has been reduced, however, some degradation in speed can result that increasing the index will not recover. Reducing the index size results in fewer "place markers," which are used by **ReadLineNumber()**. This means that typically more scanning must be done to satisfy a **ReadLineNumber()** request, which slows response time. Increasing the index size, conversely, will shorten the typical scanning time, and thus speed up response time. Although index size is in bytes, an index entry takes four bytes, so the number of lines that can be "remembered" equals the index size divided by four.

13.5.7.1 Call

A call to **SetBufferSize()**, **SetIndexSize()**, or **SetTabSize()** is made according to the following syntax:

Model

```
call { SetBufferSize( FILE_ID, BUFFER_SIZE ) |  
      SetIndexSize( FILE_ID, INDEX_SIZE ) |  
      SetTabSize( FILE_ID, TAB_SIZE ) }
```

13.5.7.2 Example

The following example of an EASEL OS/2 EE program shows the declaration of FILE_ID, BUFFER_SIZE, and INDEX_SIZE, and TAB_SIZE, and how they are used in calls to the **SetBufferSize()**, **SetIndexSize()**, and **SetTabSize()** subroutines.

```
fileid FileID
integer BufferSize is 1024
integer IndexSize is 1024

include "fileio.inc"

# To be brief, error handling is omitted

response to line "Double Buffer" from keyboard
    copy (BufferSize * 2) to BufferSize
    call SetBufferSize( FileID, BufferSize )

response to line "Halve Buffer" from keyboard
    copy (BufferSize / 2) to BufferSize
    call SetBufferSize( FileID, BufferSize )

response to line "Double Index Size" from keyboard
    copy (IndexSize * 2) to IndexSize
    call SetIndexSize( FileID, IndexSize )

response to line "Halve Index Size" from keyboard
    copy (IndexSize / 2) to IndexSize
    call SetIndexSize( FileID, IndexSize )

response to line "Set Tab Size" from keyboard
    call SetTabSize( FileID, TabSize )
```

13.5.7.3 Parameter Adjustment

The following errors can be returned by any of the Parameter Adjustment subroutines.

13.5.7.4 Subroutines Error Messages

errorlevel	Error Message/Explanation
1	Error locking global memory. Indicates a serious problem. Contact your support representative.
2	File has not been opened. It is an error to adjust a file that has not been opened.

Problem reallocating buffer (insufficient memory).

There is insufficient memory to set buffer or index to the size requested.

13.5.8 Error Handling Subroutine

If a File I/O Library subroutine encounters an error, processing is halted and the error level is set to the number of the error encountered. The value can be retrieved through the built-in function **errorlevel**. When no errors are detected, **errorlevel** contains the value zero. **GetError()** is a subroutine which sets a string variable to the error message associated with the error level passed to it. **GetError()** is defined in the include file (**fileio.inc**) according to the following syntax:

```
subroutine GetError(integer:ERROR_LEVEL, string:MESSAGE)
  library "fileio"
```

Where: ERROR_LEVEL is an integer variable that specifies an error code. To set ERROR_LEVEL, copy the value of the built-in function **errorlevel** to it (see example below).

MESSAGE is a string variable that contains a short error message after the call to **GetError()** is made.

13.5.8.1 Call

A call to **GetError()** is made according to the following syntax:

Model

```
call GetError( ERROR_LEVEL, MESSAGE )
```

13.5.8.2 Example

The following example of an EASEL OS/2 EE program shows the declaration of ERROR_LEVEL and MESSAGE, and how they are used in a call to the **GetError()** subroutine.

```
fileid FileIDin
string FileName is "input.txt"
string Read is "read"
string Message
integer ErrorLevel
include "fileio.inc"
action ErrorHandler is
  copy errorlevel to ErrorLevel
  if (ErrorLevel > 0) then
    call GetError( ErrorLevel, Message )
    send "An error was detected, Code= " ErrorLevel
```

```
"\n Error Message: " Message to errorlog
```

```
response to start
```

```
call OpenFile( FileIDin, FileName, Read )
```

```
action ErrorHandler
```

13.6 The Message Library Function

You use the message library to create message boxes. Message boxes are a quick and easy way to send a message and receive a reply, for example, to request confirmation when the user asks to quit the EASEL OS/2 EE session. Use the message box defined below to double check that the user really does want to quit before exiting:

```
include "message.inc"

response to Quit
  if ( ReplyToMessage("Quit",
                     "Do you really want to Quit?",
                     MessageYesNo,
                     2,
                     MessageNoIcon)
      = "yes") then
    exit
  end if
```

Message boxes, unlike other EASEL OS/2 EE objects, are temporary. Instead of creating them and responding to them using the EASEL OS/2 EE language as you would a key or a region, you make just one function call that creates and displays the message box you specify. Users must respond to the message box before performing any other EASEL OS/2 EE actions (such as selecting other objects). Once EASEL OS/2 EE receives the response, it removes the message box from the screen and returns the user's response to the EASEL OS/2 EE program.

You must include a header file that declares the function and defines the constants (such as `MessageYesNo`, in the example above) that you use when you call it:

```
include "message.inc"
```

In `message.inc`, **`ReplyToMessage( )`** is declared as follows:

```
function ReplyToMessage( string:TITLE,
                        string:MESSAGE,
                        integer:OPTIONS,
                        integer:DEFAULT_OPTION,
                        integer:ICON )

  returns string
  library "message"
```

To invoke **`ReplyToMessage( )`**, use the following syntax model:

```
ReplyToMessage( TITLE,  
                MESSAGE,  
                OPTIONS,  
                DEFAULT_OPTION,  
                ICON )
```

When invoking the function, you can select different values for each argument to get a different message box each time. The first argument, **TITLE**, is the title of the message box. The second is the text of the message. **OPTIONS** describes the replies that the user can choose from. You can specify one of six sets of replies:

- MessageOK**
- MessageOKCancel**
- MessageRetryCancel**
- MessageAbortRetryIgnore**
- MessageYesNo**
- MessageYesNoCancel**

In the example above, specifying **MessageYesNo** means the message box will present the user with two options: “Yes” and “No”. These options also determine the possible return values of the function. Because **ReplyToMessage()** returns the option the user selected, the possible return values in the example are “yes” and “no”. (**Note:** If an error condition occurs, “error” may be returned instead; “error” is always a possible return value.)

The fourth argument, **DEFAULT_OPTION**, specifies which option is the default. The example specifies 2, meaning option 2: “no”.

You can select one of four symbols to appear in the message box; each symbol identifies a different type of message:

- MessageQuery**
- MessageWarning**
- MessageInformation**
- MessageCritical**

Select the symbol you want by specifying it as the fifth argument. If you do not want a symbol, specify **MessageNoIcon**.

Use these parameters to display a different confirm-quit message box:

response to Quit

```
if ( ReplyToMessage( "You have selected the Quit function.",
                    "Exiting your program...",
                    MessageOKCancel, 1, MessageWarning ) = "ok") then
    exit
end if
```

This message box has “ok” as the default rather than “no”, the second option is “cancel”, and an exclamation point appears for emphasis.

Chapter 14. Application Communications, PM Clipboard, and Stimulus

14.1 Overview of Applications

14.1.1 Application Types: Remote and Local

An EASEL OS/2 EE program can communicate with multiple local and/or remote application programs. A *local application* is an application that runs on the workstation at the same time as EASEL OS/2 EE. A *remote application* is a mechanism for communicating with a program running on another computer system that is connected via an asynchronous communication link.

In each EASEL OS/2 EE program, you define the application programs you will communicate with by specifying logical names of these applications in an **application** statement, which is an environmental definition.

You must specify a **start** statement to start the application programs before EASEL OS/2 EE can receive information from, or send information to, these applications.

You can specify a **stop** action statement to stop the applications explicitly, or you can allow an application to stop automatically when the EASEL OS/2 EE program exits. EASEL OS/2 EE communicates with application programs by sending and receiving information in the form of ASCII characters. The EASEL OS/2 EE program specifies the characters that are to be sent, using the **send** action statement.

In the **response to line** or **response to char** response statements, you specify either the exact characters or the pattern of characters to which EASEL OS/2 EE is to respond. You also specify whether the characters come from a program or from the keyboard. If they come from a program, you must specify the program name. As EASEL OS/2 EE receives input from the application (or from the keyboard), it compares this input to the characters or pattern of characters specified in the response definition. When a match is detected, EASEL OS/2 EE performs the action statements within that response definition.

All applications are global. In one EASEL OS/2 EE program, you might define an application and start it running. In the other programs to which control is transferred, you simply define the application—you do not restart it. The application continues to run without interruption as control is transferred between programs; the final EASEL OS/2 EE program which uses the application may stop it.

14.1.2 Defining, Starting, and Stopping Applications

You define, start, and stop local or remote applications with the same EASEL OS/2 EE syntax, described in this section. The only difference is in the parameters used to start the application.

14.1.2.1 Defining an Application

You must define all application programs with which your EASEL OS/2 EE program will communicate, before they can be recognized by EASEL OS/2 EE. This is done with the environment declaration, as shown in the following model. The **application** environment declaration should appear at the top of the EASEL OS/2 EE program you will use it in. The form of the **application** statement is the same whether the application is local or remote.

Model

```
application APP_NAME
```

The name that you specify in the **application** statement is a logical name and is not necessarily the name by which the operating system recognizes the program. You can specify any name that you wish. For example, suppose you are communicating with a database application program whose name is DN532.EXE. Because EASEL OS/2 EE does not recognize names containing periods, you can specify a more recognizable name, as in the following:

```
application Database
```

Then, when you start the application program, you specify a **start** statement, which establishes the connection between the EASEL OS/2 EE name, Database, and the operating system name, DN532.EXE. For example:

```
response to start
  start local Database "DN532.EXE"
```

Once an application program has been defined, you refer to it throughout your EASEL OS/2 EE program by referencing the name provided in the **application** statement; for example:

```
response to line from Database
:
```

14.1.2.2 Starting an Application

You must start all application programs before EASEL OS/2 EE can receive information from, or send information to, these programs. This is done with the **start** action statement shown in the following model.

```
start { local LOCAL_APP_NAME PROGRAM_NAME [PARAMETERS] |  
        remote REMOTE_APP_NAME PORT [CHARACTERISTICS] }
```

Like other action statements, the **start** action statement can be specified in a response definition or in an action subroutine definition.

14.1.2.3 Starting a Local Application

You start a local application by specifying the **local** keyword in the **start** statement. You identify the program (LOCAL_APP_NAME entry) by specifying the same name provided in the **application** statement that defined the application program. You specify the name by which the operating system recognizes the application (PROGRAM_NAME entry), followed by any optional parameters (PARAMETERS entry).

The parameters that you specify are passed to the local application program as soon as it is started. As a parameter, you can specify any value that the program requires; for example, you might pass file information that the program needs in order to begin processing, such as:

```
response to ...  
    start local Database "query.exe" "history.dat"
```

Note that the PROGRAM_NAME and PARAMETERS string values must be enclosed in quotation marks.

This statement starts a local application named Database. The name of this program, as recognized by the operating system, is "QUERY.EXE". When the program is started, EASEL OS/2 EE passes the parameter string, "HISTORY.DAT", to the program, which identifies the database to be queried.

Note: You can run only five local applications at a time. If you try to start a sixth without stopping any running application, an error message is generated.

14.1.2.4 Starting a Remote Application

You tell EASEL OS/2 EE that you are starting communication with a remote application by specifying the keyword **remote** in the **start** statement. You identify the program (REMOTE_APP_NAME entry) by specifying the same name specified in the **application** environment declaration that defined the application program. You must also specify one of the following keywords for the communications port (PORT entry) to be used for communication:

"com1"

"com2"

"com3"

Note that the PORT string value must be enclosed in quotation marks.

For the CHARACTERISTICS entry, you can optionally specify a string value representing the characteristics of the communications port. These options and their defaults are explained in 14.3.4, "ACDI Remote Characteristics Keywords" on page 14-21.

The CHARACTERISTICS string value must also be enclosed in quotation marks. If more than one characteristic is required, separate each characteristic with a space. The characteristics can appear in any order within the string. For example:

response to ...

```
start remote Sales "com1" "1200 evenpar"
```

will open the communications channel with the remote application Sales via serial port COM1, at 1200 baud, generating even parity.

If you do not specify characteristics, EASEL OS/2 EE uses the defaults specified in the CONFIG.ESL file by the REMOTE keyword. If you specify characteristics, EASEL OS/2 EE automatically appends them to any characteristics specified by REMOTE. Keywords appearing later in the resulting characteristics string override any conflicting keywords that appear earlier in the string.

14.1.2.5 Stopping an Application

The **stop** action statement stops a local application program and communications with that program, or stops communications with a remote application program.

Model

```
stop { APPLICATION_NAME | all }
```

You can stop a single application program; the identifier specified as the application name must be the name of an application program defined in an **application** statement. For example:

```
application Prog
:
response to ...
    stop Prog
```

or you can specify **stop all**, which stops all local applications and stops communication with all remote applications. For example:

```
response to ...
    stop all
```

The **stop** statement is optional. If you do not stop an application, EASEL OS/2 EE stops it automatically when an **exit** statement is executed. Once an application is stopped, no further responses are taken for that application, even if the application has already sent input to the EASEL OS/2 EE program.

A local application that is stopped is terminated immediately, even if there is pending processing for that application (for example, even if EASEL OS/2 EE has sent output that the application has not yet processed).

14.2 Sending and Receiving Data

Data is sent to and received from local or remote applications in the same way, with the EASEL OS/2 EE statements described in this section.

14.2.1 Sending Information

The **send** statement, shown in the model below, is used to send information to an application or to the errorlog. (The errorlog is used for debugging.)

Model

```
send VALUE1 [VALUE2]... to { APP_NAME | errorlog }
```

As the APP_NAME identifier, you specify the same name of the application that you defined in the **application** environment declaration. Remember also that an application must be started with the **start** statement before you can send information to it.

You can send any number of values to an application program or to the errorlog in a single **send** statement.

14.2.1.1 Escape Sequences in Applications

When you send data to an application program, you often must end in the data with the newline escape sequence, “\r”, “\r\n” for most remote applications and “\n” for local applications. This sequence simulates the RETURN/ENTER key on the keyboard. Most applications do not recognize input until they receive the newline character. For example, the statement:

```
send "Start\n" to Database
```

sends the word, “Start,” terminated by a newline character, to the application program named Database.

14.2.2 Responding to Data Received from an Application

The **response to line** or **response to char** statement lets you respond to remote or local applications, or to the keyboard. You specify the **response to line** or **response to char** statement in a response definition that is invoked when EASEL OS/2 EE recognizes text. The text might be entered by the user on the keyboard or sent to EASEL OS/2 EE by an application program.

Note that you should specify **response to line** rather than **response to char** whenever possible, to increase processing efficiency. Only use **response to char** when EASEL OS/2 EE needs to respond to a prompt (that does not end with the newline character). The **response to char** statement is used most efficiently when in a block structure. See 6.2, "Working with Blocks" on page 6-10 for more information on blocks.

14.2.2.1 Responding to Line Input

The **response to line** statement tells EASEL OS/2 EE that the specified response is to be triggered when a complete line of input is received from the keyboard or from a local or remote application program. EASEL OS/2 EE considers a complete line to be any stream of characters that ends with the newline (line feed) character. This statement uses the following model.

Model

```
[interrupt] response to line [MATCH_CLAUSE]...  
  from { keyboard | APP_NAME }  
  
  [ACTION_STATEMENT]...
```

You can specify, as the MATCH_CLAUSE, one or more match clauses showing the format or composition that a line must have before it can trigger the response. For example:

```
response to line "+" from Database  
:
```

Here, the character "+" must be received within a line of text (terminated by an \n) from the application program for the response to be triggered.

Match clauses are described in 14.2.2.3, "Match Clauses for Line or Character Input" on page 14-9.

As shown in the model, the match clause is optional; for example:

```
response to line from Database  
:
```

Here, if any line (or even just a newline character) is received from the application Database, the response will be taken.

14.2.2.2 Responding to Character Input

The **response to char** statement tells EASEL OS/2 EE that the specified response is to be triggered when specified character(s) are received from the keyboard or from a local or remote application program. The input need not end with a newline character. This statement uses the following model.

Model

```
[interrupt] response to char [MATCH_CLAUSE]...  
    from { keyboard | APP_NAME }  
  
    [ACTION_STATEMENT]...
```

You can specify, as the MATCH_CLAUSE, one or more match clauses showing the format or composition of the input that EASEL OS/2 EE is to recognize. For example:

```
response to char "Please enter name:" from Prog_1  
    :
```

Here, the precise character string "Please enter name:" must be received from the application program for the response to be triggered. As shown in the model, the match clause is optional. For example:

```
response to char from Mail  
    :
```

Here, if *any* character is received from the application named Mail, the response will be taken.

Match clauses are described in 14.2.2.3, "Match Clauses for Line or Character Input."

14.2.2.3 Match Clauses for Line or Character Input

Match clauses may be specified for the **response to line** or **response to char** response statements. The MATCH_CLAUSE, in the **response to line** and **response to char** statements, uses the following model.

Model

```
[col NUM1 [thru NUM2] ] { STRING | word | number | blank }
```

You can specify any number of match clauses in a **response to line** or **response to char** statement. The response will be invoked when the input

received by EASEL OS/2 EE matches every match clause specified for that response.

The match clause can specify a particular character string, a blank, any word, any number, or any combination of these. Each match clause must be a compile-time string value.

14.2.2.4 Column Positions in Match Clauses

You can optionally specify column positions in which the input must appear. The column positions can be exact, for example:

```
col 2 "N"
```

This is satisfied if the character "N" appears in column position 2. Or the column positions can be approximate:

```
col 2 thru 7 "N"
```

This is satisfied if the character "N" appears anywhere in column positions 2 through 7.

14.2.2.5 STRING in Match Clauses

The match clause can indicate a specific character string that must appear in the input. The string must be a compile-time value. You can define the exact column positions in which the string must appear, a range of column positions in which the string must appear, or no column positions, so that the presence of the string anywhere in the input will satisfy the match clause.

For example, the response:

```
response to line "No diagnostics" from Prog
```

is taken if EASEL OS/2 EE receives a line containing the text "No diagnostics" from the application named Prog. The following response:

```
response to char col 1 "READY>" from Prog
```

is taken if the character "R" in the string "READY> " appears starting in column position 1 of the input. Or, for example:

```
response to char col 1 thru 8 "READY>"
```

requires that *all* of the characters in the string "READY> " appear anywhere in positions 1 through 8. This match clause is satisfied if any of the following column positions contain "READY>."

1-6

2-7

3-8

The string must be contiguous; "R" in column position 2 and "EADY>" in column positions 4 through 8 do not match, because column position 3 contains a character that is not part of the specified string. In the following match clause:

```
response to char "READY>"
```

a match is found if the contiguous characters, "READY>", appear anywhere in the input.

You can use special characters received from the keyboard in match clauses. These characters are described in 15.4, "Handling Characters Received from the Keyboard" on page 15-9.

14.2.2.6 word in Match Clauses

A match clause can indicate that a word must appear in the input. A word is identified by the keyword **word** in the match clause. A word is any contiguous group of ASCII extended characters, including digits but not including blanks, control characters, and delete characters. As with a character string, you can specify the exact positions that must contain a word, a range of positions, or no positions. In the response statement:

```
response to char col 6 word from Prog
```

column 6 must begin a word. Because this match clause does not specify an ending position, the word may be of any length, including a single character. In the following example, a word must appear somewhere in positions 6 through 10:

```
response to char col 6 thru 10 word from Prog
```

Any printable character in columns 6, 7, 8, 9, or 10 results in a match.

14.2.2.7 number in Match Clauses

A match clause can indicate that a number must appear in the input. A number is identified by the keyword **number** in a match clause. A number is an integer, consisting of any digits, optionally preceded by a sign (+ or -) or leading blanks, and without embedded commas or decimal points. (Note that floating point values cannot be taken.) For example:

```
response to char col 11 number from Prog
```

specifies that position 11 must contain an integer. Because there is no ending position in this match clause, it is satisfied by a number of any length.

In the match clause for the following response statement:

```
response to char col 11 thru 16 number from Prog
```

there must be a number in positions 11 through 16, beginning in position 11. The number can have leading blanks or a sign, but no other characters may precede the number. Any of the following, beginning in column 11, satisfy this match clause:

12
12
12X

The character strings below do not satisfy this match clause:

X12
- M123

Characters that match a number specification must contain at least one digit. For example, the following:

- 42

in columns 1 through 4 does not satisfy a response statement with a match clause such as the one below:

response to char col 1 thru 2 number from Prog

EASEL OS/2 EE finds only the minus sign and leading blank (" - ") in columns 1 and 2. Although these characters can be part of a valid number, they do not, by themselves, satisfy a match clause that specifies a number.

Embedded blanks, commas, and decimal points are not allowed in numbers. For example, the number:

.35

in column positions 3 through 5 does not satisfy a response statement with a match clause such as the following:

response to char col 3 thru 5 number from Prog

However, this match clause is satisfied if the number:

123.5

begins in column 3, because "123" is a valid number.

14.2.2.8 blank in Match Clauses

A match clause can indicate that a blank character must appear in the input. A blank is identified by the keyword **blank** in the match clause. You can define the exact column position that must be blank, the range of column positions (one of which must contain a blank), or no column positions, so that the presence of a blank anywhere in the input satisfies the match clause. For example:

response to line col 9 blank from Prog

requires that column position 9 of the input be blank and that the input be terminated with a newline character (\n). The presence of any nonblank character in this position does not satisfy the match clause. In the response statement:

response to char col 9 thru 15 blank from Prog

there must be a blank character somewhere within positions 9 through 15 of the input. The response statement:

response to char blank from Prog

is satisfied if any column position in the input is blank.

14.2.2.9 Multiple Match Clauses

You can specify more than one match clause in a single **response to char** or **response to line** response statement, for example:

```
response to line
  col 1 "add"
  col 5 thru 10 number
  from application
  :
```

This response statement contains two match clauses. The first match clause:

```
col 1 "add"
```

is satisfied if the characters "add" appear in column positions 1-3 of the input line. The second match clause:

```
col 5 thru 10 number
```

is satisfied if column positions 5 through 10 contain a valid number.

Both match clauses must be satisfied in order for the conditions specified in the response statement to be met. If either match clause is not matched by the input line, the response statement is not satisfied and EASEL OS/2 EE does not perform the action statements specified in the response definition.

14.2.3 How response to line and response to char Are Processed

When a MATCH_CLAUSE is satisfied and the response definition is invoked, all characters up to and including the matched pattern are removed from the input stream and copied to the built-in function **input**. Further pattern matching will begin at the next character (any character, including newline, special character, blank, and so on) in the input stream; this character position becomes column 1 of the input stream.

In a **response to line** statement, characters are accepted and are buffered until a newline character is encountered. At this point, EASEL OS/2 EE will look for a match in the response definition. If no match is found, the buffer is cleared up to that point. If a match is found, the **input** function returns the entire line, and the buffer is cleared up to that point. The next line that EASEL OS/2 EE examines for a match begins with the first column position following the newline character.

In a **response to char** response statement, characters are accepted and examined upon receipt for a match. If no match is found, the character is buffered until a newline character is encountered, at which point the buffer is cleared up to the newline character, and EASEL OS/2 EE begins further examination at the next column position, which is now called column position 1. If a match is found, the **input** function returns the characters in the buffer up to and including the last character of the match clause, and the buffer will be cleared up to that point. For example:

```
response to char "end" from Prog
:
response to char col 9 thru 11 "32a" from Prog
:
```

Assume that the input stream from Prog is the following:

```
This is the end \nof file 32a\n
```

When the first response is taken, the characters "This is the end" are copied to **input**. When the second response is taken, the characters "of file 32a" are copied to **input**.

14.2.3.1 Extracting Data Fields from Character Input

The **extract from** action statement, described in Chapter 4, allows from you to extract data fields from the character input that is received from an application program or the keyboard and is returned by the **input** function. You can store these data fields in variables.

14.2.3.2 Block Structure Processing

When EASEL OS/2 EE receives text from an application program or the keyboard, it searches for a valid **response to char** or **response to line** response definition. EASEL OS/2 EE first searches in the innermost active block. If blocks are not specified in the program, EASEL OS/2 EE searches the entire program and takes the first valid response definition.

If the block contains a valid response definition for the input, that response is taken immediately.

Note: If more than one such response definition is found, the first valid response definition is invoked—even if there are other responses earlier in

the block that would be satisfied if more of the input were received before responding.

Consider the following example:

```
begin
  response to line from keyboard
  :
  response to char "AB" from keyboard
  :
  response to char "A" from keyboard
  :
end
```

If the input from the keyboard consists of the string:

"AB\n"

then EASEL OS/2 EE takes the third response:

```
response to char "A"
:
```

as soon as it receives the "A", without waiting to see if anything follows.
When EASEL OS/2 EE receives the rest of the line:

"B\n"

it takes the first response:

```
response to line
```

Note that the effect described above is observed only within the innermost active block. Consider the following:

```
begin BlockA
  response to BlueKey
  begin BlockB
    response to line from App
  :
    response to char "AB" from App
  :
  end BlockB
  response to char "A" from App
  :
end BlockA
```

In this example assume that you are in Block B. No response is taken when EASEL OS/2 EE receives the "A". When EASEL OS/2 EE receives the "B," it takes the second response:

```
response to char "AB" from App
```

EASEL OS/2 EE attempts to satisfy the innermost block until it has recognized enough input to determine that none of the inner responses can be satisfied. In the example above, suppose that the **response to line** statement was missing and the input was "AC\n." Because EASEL OS/2 EE has determined that there is no "AB" in the line, it takes the outer response, matching the "A".

If EASEL OS/2 EE does not find a match between any response definition statement and the current input, even after searching all active blocks, the current input is discarded. If EASEL OS/2 EE does find a match, it performs the action statements specified in the response definition.

14.2.3.3 Using response to char in Blocks

When you use **response to char** response statement in a block, be careful to ensure that EASEL OS/2 EE knows when no more information is expected, so that all available responses—including those outside the block—can be taken. In order to check for a pattern match in an outer block, the pattern in the inner block must not be matched. When you are using **response to char**, the *only* way to do this is by using a specific pattern match that excludes any other input. Use the **col** specification in the MATCH_CLAUSE to make sure that all available responses can be taken.

14.2.4 Changing the Disposition of Text Input

One way of controlling EASEL OS/2 EE's responses to input was described in 14.2.3, "How response to line and response to char Are Processed" on page 14-13. You can also control EASEL OS/2 EE's processing of responses to input by using the **make string disposition** action statement, as shown in the following model.

Model

```
make string disposition { keyboard | APPLICATION_NAME }
  { normal | drop | hold }
```

As shown in the model, you can change the disposition of input from the keyboard or from a local or remote application program, and you can select one of the following dispositions:

Disposition	Action Taken
normal	When EASEL OS/2 EE receives input, it performs the appropriate response. This is the default disposition.
drop	When EASEL OS/2 EE receives input, it discards the input.

hold

When EASEL OS/2 EE receives input, it saves the input. When the disposition becomes **normal** again, EASEL OS/2 EE handles the saved inputs, using the appropriate responses.

14.3 Remote Application Support for EASEL OS/2 EE

EASEL OS/2 EE uses the Communications Manager (CM) Asynchronous Communications Device Interface (ACDI) for remote application communications.

ACDI has the following installation requirements:

- CM must be installed
- Device driver C:\CMLIB\ASYNCDx.SYS (x = A or B) must be installed via CONFIG.SYS for the serial devices (for example, COM1, COM2, and so on) to be used
- CM must be configured for asynchronous communications

See the IBM OS/2 EE manuals for installation and configuration information for the above items.

14.3.1 Starting Remote Applications

Before EASEL OS/2 EE can start a remote application, the following conditions must apply:

- CM has been started
- ACDI has been started, either via CM menu selections or automatically, when CM is started (optional)

Use the **start remote** action statement to start the application.

Model

```
start remote REMOTE_APP_NAME PORT [CHARACTERISTICS]
```

14.3.2 Connection Types

You can specify one of five connection types when you start a remote application:

Connection Type	Characteristics Keyword
Auto Dial	autodial
Auto Answer	autoanswer
Manual Dial	manualdial
Manual Answer	manualanswer

No command set **direct**
(user-established)

For all connection types except "No command set," a modem must be attached to the port to be used for the remote application. Supported modem types include all those for which CM can be configured, including various models of IBM and Hayes modems. The "No command set" (referred to from now on as "Direct") connection type is used if there is no modem attached, for example, when the port is connected directly to the computer, or when you are using a modem that is not supported by CM. The default connection type is Direct.

For any connection type, the connection is attempted when the **start remote** action statement is executed. Except for the Direct connection type, the connection attempt will usually still be in progress after the **start remote** action statement has completed. This allows EASEL OS/2 EE to continue processing other actions or user input while waiting for the connection to be made.

14.3.2.1 Auto Dial

Auto Dial causes ACDI to send commands to the modem, which dials a number and establishes a connection. You must specify the number to be dialed using the **number** keyword followed by the telephone number. You can also use the **prefix** and **suffix** keywords to dial additional digits before and after the main number, to meet the requirements of various telephone networks. For example, you might specify "**prefix 9**" if your office telephone network requires you to dial 9 to access an outside line.

Put a comma within the string argument of any of the keywords **prefix**, **number**, and **suffix** if you want the modem to pause for one second before continuing to process your connection.

Specify multiple commas to create longer pauses. You can choose a different character as the pause character by including the **pausechar** keyword in your characteristics string.

14.3.2.2 Auto Answer

Auto Answer causes ACDI to automatically wait for and respond to an incoming phone call.

14.3.2.3 Manual Dial

Manual Dial mode is typically used with an acoustic coupler type of modem. The user dials the number, gets to the point where the computer tone is heard, then selects a control on the application, which issues a **start remote** action, and the connection is completed.

14.3.2.4 Manual Answer

In Manual Answer mode, the user answers the telephone, places it in the acoustic coupler, then selects a control that issues the **start remote** action.

14.3.3 Establishing a Connection

Use one of the following methods to establish a connection:

14.3.3.1 Direct Connection No Timeout

For a direct connection mode with no timeout specified, use a statement similar to the following:

```
start remote APPL1 "COM1" "direct"
```

As soon as **start** returns, the connection is established.

14.3.3.2 All Other Modes and Direct Connection with Timeout

For all the following modes:

- Auto Dial
- Auto Answer
- Manual Dial
- Manual Answer
- Direct, if a timeout is specified

first start the application with a statement similar to one of the following:

```
start remote APPL1 "COM1" "autodial 12223456"  
start remote APPL1 "COM1" "autoanswer"  
start remote APPL1 "COM1" "manualdial"  
start remote APPL1 "COM1" "manualanswer"  
start remote APPL1 "COM1" "direct timeout 2"
```

You must have response statements active to recognize a successful or failed connection attempt. A successful connection returns the message line
:CONNECT

An unsuccessful attempt returns the message line

```
:FAIL n
```

where n is the failure code listed in 14.3.5, "Failure Codes" on page 14-24.

EASEL OS/2 EE—not ACDI, the modem, or the remote application program—generates these messages. You are certain to get one of the two messages before any other input.

You should place the response statements inside a guarded block following the **start remote** statement, as follows:

```
response to DialIt
  start remote Bulletin "com2" " 2400 autodial
  number 15552223456"
  begin guarded
    response to line ":CONNECT" from Bulletin
    leave block
    response to line ":FAIL" from Bulletin
    stop Bulletin
    send "Cannot connect to Bulletin Board\n"
    to ...
  end
```

Note: You must still use the **stop** action statement to stop a remote application when a **:FAIL** is received.

After a successful connection is made, the application input buffer will still contain the dialog between ACDI and the modem that established the connection. Because the EASEL OS/2 EE program does not need or use this input, you should write responses that are activated, not by the modem dialog messages, but by the first characters sent by the remote application program itself.

14.3.4 ACDI Remote Characteristics Keywords

You can use the following parameters as part of the **start remote** syntax.

14.3.4.1 Flow Control

Characteristic	Tells EASEL OS/2 EE To...
tandem (default)	Send and obey flow control
-tandem	Do not send or obey flow control

14.3.4.2 Parity

Characteristic	Tells EASEL OS/2 EE To...
nopar (default)	Do not receive or transmit parity
evenpar	Generate even parity
oddpair	Generate odd parity
markpar	Generate mark parity (parity bit = 1)
spacepar	Generate space parity (parity bit = 0)

14.3.4.3 Databits

Characteristic	Tells EASEL OS/2 EE To...
databits 7	Generate 7 bits per character (this is the default, if parity is anything except nopar)
databits 8	Generate 8 bits per character (this is the default, if parity is nopar)

14.3.4.4 Stopbits

Characteristic	Tells EASEL OS/2 EE To...
stopbits 1 (default)	Generate one stop bit per character
stopbits 2	Generate two stop bits per character

14.3.4.5 Baud Rates

You can specify any of the following baud rates:

110
150
300
600
1200
2400
4800
9600
19200

14.3.4.6 I/O Buffers

The following parameters set the size of the input and output buffers used for communications. N defines the number of characters (bytes) the buffer can hold and must be an integer between 128 and 65535.

Model	Default
inbufsize N	N = 1024
outbufsize N	N = 128

14.3.4.7 Flow Control Thresholds

The following parameters set the flow control threshold levels used to determine when XON and XOFF characters need to be sent to the application. The value of N defines the number of unused bytes in the input buffer. If flow control is enabled (via the **tandem** keyword), an XOFF character is sent when the number of unused bytes in the input buffer decreases to the XOFF

threshold value, and an XON character is sent when the amount of empty space in the input buffer increases to the XON threshold value.

The XON threshold value must be greater than the XOFF threshold value but no greater than the input buffer size. You should space the two thresholds far enough apart so that XON and XOFF characters do not need to be sent too often.

Model	Default
xonthresh N	N = 80% of input buffer size
xoffthresh N	N = 20% of input buffer size

14.3.4.8 XON/XOFF Flow Control

The following parameters specify the control characters used for XON/XOFF flow control. The default settings are nearly always used by host computers for XON/XOFF flow control; you are unlikely to need these keywords in your remote characteristics string.

Model	Default
xonchar CONTROL_CHAR	xonchar \^Q
xoffchar CONTROL_CHAR	xoffchar \^S

14.3.4.9 Timeout Value

The keyword **timeout** specifies a timeout value in seconds. The message line

" :FAIL 103"

is returned if a connection cannot be made within the specified number of seconds. The timeout value must be a number between 1 and 65536, except for AutoDial and AutoAnswer, where a value of zero has special meaning.

Model	Optional/Required
timeout SECONDS	Optional

Default timeout is as follows:

Connection Type	Default
Direct	1 second
Manual Dial	30 seconds
Manual Answer	30 seconds
Auto Dial	0 (ACDI supplies timeout)
Auto Answer	0 (timeout is infinite)

14.3.4.10 Auto Dial Access Codes

Use the keywords below when Auto Dial is specified. See 14.3.2.1, “Auto Dial” on page 14-19 for more information about the Auto Dial connection type.

Model	Optional/Required
number TELENET_NUMBER_STRING	Required
prefix TELENET_PREFIX_STRING	Optional (Default: null prefix string)
suffix TELENET_SUFFIX_STRING	Optional (Default: null suffix string)
pausechar PAUSE_CHARACTER	Optional (Default: , (comma))

14.3.5 Failure Codes

The following are the connection failure codes and their meanings:

Code	Error Description
97	DCE hardware not responding
99	Data Set Ready (DSR) signal was not negated for at least 100 milliseconds
101	Call violates country requirements
102	Call attempt cannot be made
103	Timeout occurred
105	Physical port being used for a connection
108	Requested connection timeout value violates network connection requirement
113	Connection type conflicts with CM serial port modem configuration
123	Call collision occurred
241	Base operating system error occurred

14.3.6 Errorlevel Built-in Function

You can use the **errorlevel** built-in function to determine if an application (remote or local) has been successfully started. You can also use it to determine whether an error occurred while sending data to started application.

Example 1

```
start remote Host "COM2"  
    "1200 databits 7 stopbits 1 evenpar"  
if (errorlevel != 0) then  
    # Application could not be started  
    exit  
end if
```

Example 2

```
send "display rpt14\r" to Host  
if (errorlevel = 0) then  
    # Command was successfully sent  
    action RetrieveReport  
else  
    # Command could not be sent  
    make IO_ErrorMessage visible  
    stop Host  
end if
```

You should test only for errorlevel being equal to or not equal to zero, where zero indicates success and non-zero indicates failure.

14.4 Sample Programs

14.4.1 Sample Program Using a Local and Remote Application

screen size 640 350

application CapAll

application Host

Graphical Object Definitions:

blue region Password size 400 200

at position 70 75

white border foreground

red key Exit at position 400 10

solid box 100 80

white box 100 80

move to 50 40

white text center "EXIT"

Response Definitions:

response to start

start local CappAll "caps.exe"

start remote Host "COM1"

"1200 autodial number 15552223456"

begin guarded

response to line " :CONNECT" from Host

leave block

response to line " :FAIL" from Host

send "ERROR CONNECTING TO REMOTE APPLICATION ^n"

to errorlog

stop all

exit

end

response to line from keyboard

send input "\r\n" to CapAll

response to line from CapAll

change Password to

move to xmiddle ymiddle

font "asc1521" text center input

send input "\n" to Host

```
response to Exit
    stop all
    exit
```

14.4.2 Sample Program Using Characters Received from the Keyboard

```
# Each character that is typed by the user is displayed
# on the screen. The line of displayed text grows,
# character-by-character, until the user ends the line
# with a carriage return or deletes the line.
```

```
string variable CompleteLine
textual region TextEcho size 800 30
    at position 0 200
```

```
# If the user deletes the current line of text by
# simultaneously pressing the CONTROL and the U keys, the
# line disappears from the display screen:
```

```
response to char "\^U" from keyboard
    copy "" to CompleteLine
    clear TextEcho
```

```
# When the user completes a line by pressing the return
# key, the specified actions are taken and the displayed
# line disappears from the screen:
```

```
response to char "\n" from keyboard
    copy textual line 1 from TextEcho to CompleteLine
    clear TextEcho
```

```
response to char from keyboard
    add to TextEcho
    insert input
```

14.5 Using the Presentation Manager Clipboard

The PM clipboard is a general storage area that you can use to exchange data between PM applications. The exchange of data is usually user controlled and initiated from functions in the application such as cut, paste, and copy. EASEL OS/2 EE allows you to copy textual information to and retrieve it from the clipboard. You should use the clipboard to implement textual functions such as cut, paste, and copy, for the user.

To copy text to the clipboard from EASEL OS/2 EE use the **copy to clipboard** statement.

Model

copy STRING to clipboard

You can copy any EASEL OS/2 EE string value to the clipboard. After the copy is executed, any textual data that previously existed in the clipboard is entirely replaced by the string value.

Note: You cannot append a string to the clipboard.

Use the built-in function **clipboard** to access the textual data in the clipboard.

Model

clipboard

You can use the **clipboard** built-in function anywhere a string value is allowed in EASEL OS/2 EE. It returns the entire textual contents of the **clipboard**.

Example:

```
# Copy clipboard contents to a textual region.  
add to PasteRegion  
    insert clipboard
```

14.6 Stimulus Libraries

The stimulus feature provides a means for a Dynamic Link Library (DLL) to cause an event in the EASEL OS/2 EE program that calls it. (Dynamic Link Libraries are discussed in 25.3, "Writing a Stimulus Library" on page 25-18.) For instance, a DLL that provides an interface to a hardware device might need to alert an EASEL OS/2 EE program when a certain condition arises on the device.

14.6.1 Declaring a Stimulus

The declaration of a stimulus links a given logical name with the name of the DLL that implements that stimulus. A stimulus must be declared before it can be referenced in a response. The following is the model for a **stimulus** declaration.

Model

```
stimulus STIMULUS_NAME DLL_NAME
```

STIMULUS_NAME is an identifier for the stimulus, and DLL_NAME is a string value identifying the name of the DLL for this stimulus. For example:

```
stimulus StimulusName "mystim"
```

14.6.2 Defining a Stimulus Response

A stimulus response specifies the action statements that EASEL OS/2 EE is to perform when the stimulus library generates an event. When the library generates an event, it provides two parameters: an event number and an event parameter. These are integer values that the library can use to provide information regarding the event. A stimulus response can optionally specify a value that must match the event number or the event parameter in order for the response to be performed. The following is the model for a **response to stimulus** definition.

Model

```
[interrupt] response to stimulus STIMULUS_NAME  
[ _ | EVENT_FILTER ] [ _ | PARM_FILTER ]  
  
[ACTION_STATEMENT]...
```

EVENT_FILTER is an integer constant or variable. In order for the response to be invoked, this value must match the event number provided by the DLL when it invokes the response. If * is specified, any event number will match.

PARAM_FILTER is an integer constant or variable. In order for the response to be invoked, this value must match the event parameter provided by the DLL when it invokes the response. If * is specified, any event parameter will match.

When searching for a matching response to a stimulus event, EASEL OS/2 EE executes the first response that matches the given event number and parameter. EASEL OS/2 EE searches all currently active blocks for a matching response, until either the outermost block has been searched or a guarded block has been reached. If a matching response is not found, no response is executed and the event is discarded.

14.6.2.1 Examples

```
# This response will be executed only when eventnumber = F1 and
# eventparam = F2.
response to stimulus StimulusName F1 F2
```

```
# This response will be eventnumber = F1.
response to stimulus StimulusName F1
```

```
# This response will be eventparam = F2.
response to stimulus StimulusName * F2
```

```
# This matches any stimulus event.
response to stimulus StimulusName
```

14.6.3 Response Inquiry Functions

Two built-in response inquiry functions are available for use in a stimulus response.

Function	Value Returned
eventnumber	An integer value representing the event number provided by the DLL invoking the response. The meaning of this value is dependent on the DLL.
eventparam	An integer value representing the event parameter provided by the DLL invoking the response. The meaning of this value is dependent on the DLL and the eventnumber .

Chapter 15. Controlling the Keyboard and Pointing Devices

15.1 About Pointing Devices

EASEL OS/2 EE supports the mouse and the touchscreen as pointing devices.

The user invokes responses by using a mouse to point to graphical objects on the display. With a touchscreen, the user touches the object on a touchscreen.

Both pointing devices are global. If you open a pointing device in one program, the device remains open when control is transferred to another program (with the **change to program** statement). You do not need to reopen the device.

15.1.1 Mouse

When using a mouse, a Presentation Manager (PM) mouse pointer is visible on the display. To point to an object with a mouse, you move the mouse until the pointer is at the place on the display that you want to touch. Press the left mouse button to invoke a response to the object beneath the pointer.

15.1.2 Touchscreen

In systems with a touchscreen, the surface of the monitor is sensitive to touch. Touch the object on the monitor to invoke a response to a graphical object.

15.1.3 Mouse Pointer Appearance

The normal appearance of the mouse pointer is the standard PM slanted arrow. You can use the action statement **set pointer to** to change its appearance.

Model

```
set pointer to POINTER_NUMBER [in DLL_FILE]
```

If no file is specified, POINTER_NUMBER indicates one of the standard PM pointers shown in the following table:

15.1.3.1 Presentation Manager System Pointers

Constant Name	Pointer Description
SPTR_ARROW	Arrow cursor
SPTR_TEXT	Text entry I-beam
SPTR_WAIT	Hourglass
SPTR_MOVE	Four-way arrow
SPTR_SIZENWSE	Size border arrow for UL and LR corners
SPTR_SIZENESW	Size border arrow for LL and UR corners
SPTR_SIZEWE	Size border arrow for left and right sides
SPTR_SIZENS	Size border arrow for top and bottom
SPTR_APPICON	Large rectangle
SPTR_ICONINFORMATION	Hand inside a stop sign
SPTR_ICONQUESTION	Question mark inside inverted triangle
SPTR_ICONERROR	Exclamation mark inside a triangle
SPTR_ICONWARNING	Large asterisk inside a box
SPTR_ILLEGAL	Diagonal line inside a circle
SPTR_FILE	Rectangle with UR corner folded down
SPTR_FOLDER	File folder symbol
SPTR_MULTIFILE	Multiple overlapping rectangles
SPTR_PROGRAM	Rectangle, bold on top

The constants in this table are defined in the include file PMPTR.INC.

If a dynamic link library file is specified then the number indicates which pointer in the file should be used.

15.1.4 Touchscreen Support

EASEL OS/2 EE touchscreen support is provided by a PM application whose filename is TOUCHSCR.EXE. This program is meant to be started at system bootup time. While running, its window is always minimized; the icon appears as a hand with a finger pointing at a touchscreen monitor.

15.1.4.1 Limitations

Only Interactive Images EASELtouch® touchscreen hardware is supported by this application.

Only EASEL OS/2 EE interfaces can use the touchscreen. The touchscreen does not invoke **on activate** responses.

EASELtouch is a registered trademark of Interactive Images, Inc.

The touchscreen does not work with PM objects, such as scroll bars, sizing borders, action bars, and so on. You must use the mouse to select and control PM objects.

15.1.4.2 Creating STARTUP.CMD

If you want to have TOUCHSCR.EXE started automatically when your system is booted, create a file named STARTUP.CMD in the root directory of your boot drive (usually C:\), and place the following commands in it:

```
cd \easel-ee
start /PM touchscr
c:\easel-ee\caldata.dat
exit
```

If a STARTUP.CMD already exists, append the above commands to that file. However, make sure there is only one exit command in the file and that it is the last command.

The next time your system boots, a temporary PM window will be created in which the STARTUP.CMD commands will be run. The start /PM touchscr command starts TOUCHSCR.EXE. The exit command exits the command processor and closes the temporary window.

15.1.4.3 Adding to a Desktop Manager Group

It is also a good idea to add TOUCHSCR.EXE to a Desktop Manager Group window so that if you choose to stop the program, you can easily restart it without having to reboot your system. Calibrating the touchscreen requires that you stop TOUCHSCR.EXE; see 15.1.4.7, “Calibrating the Touchscreen” on page 15-5.

Select the Program item from the Desktop Manager Group action bar. Select New... from the pulldown and a dialog box appears. Enter the following bold text in the text entry fields:

Required

Program title:	Touch Screen
Path and file name:	c:\easel-ee\touchscr.exe

Optional:

Parameters:	c:\easel-ee\caldata.dat
-------------	--------------------------------

You can leave the “Working directory:” field blank. Select the Add push button. The dialog box disappears and a new item appears in the Desktop Manager Group list for the touchscreen program.

15.1.4.4 Starting

To start TOUCHSCR.EXE manually, go to the appropriate Desktop Manager Group window and select "Touch Screen" from the list. The icon appears as a hand with a finger pointing at a touchscreen monitor.

15.1.4.5 Stopping

You can exit TOUCHSCR.EXE at any time by selecting its icon. A system menu will appear just above the icon. Select the "Close" choice in the system menu.

You can also use the Task Manager window to close TOUCHSCR.EXE.

15.1.4.6 Setting the Default Pointer

If you want the touchscreen to be the default pointing device used by EASEL OS/2 EE, you must modify the POINTER option in your CONFIG.ESL file to read:

```
POINTER=touchscreen
```

15.1.4.7 Calibrating the Touchscreen

To calibrate the touchscreen, follow these steps:

1. TOUCHSCR.EXE must not be running while you are performing the calibration procedure. To check this, switch to the PM screen group and look for the pointing finger icon. If you see the icon, stop TOUCHSCR.EXE before running TSCAL.
2. Position yourself in front of the monitor as you will be when you use the touchscreen. For example, if you will be sitting directly in front of the monitor when you use it, sit directly in front of the monitor to calibrate the touchscreen.
3. At the fullscreen OS/2 command prompt, change to the C:\EASEL-EE directory, or whichever directory EASEL OS/2 EE was installed on, and enter the following command:

```
tscal
```

You are prompted to touch an arrow or cross on the screen whenever it changes appearance or color, or changes position on the screen. At each location you must touch the target three times before it moves to another location.

Calibration is complete when you are returned to the fullscreen OS/2 command prompt.

4. After calibration completes, restart TOUCHSCR.EXE using the appropriate Desktop Manager Group window.

15.2 Controlling Availability of Pointing Devices

15.2.1 Using the Default Pointing Device

Each EASEL OS/2 EE system has a default pointing device, usually the mouse. The default device is specified when you configure EASEL OS/2 EE; to change the default device, simply reconfigure. When EASEL OS/2 EE begins operation, the default pointing device is open and ready to use; you need not explicitly open or close it.

You can open either or both pointing devices in the same EASEL OS/2 EE program. If the mouse or touchscreen is not your default pointing device, you must open it using the **open** statement.

15.3 Opening and Closing a Pointing Device

The **open** statement activates a mouse or touchscreen as a pointing device. (If one of these devices is your system's default device, it is already activated.)

Model

```
{ open | close } { mouse | touchscreen }
```

Usually, **open** is part of a **response to start** response definition. For example:

```
response to start
  open touchscreen
```

The **close** statement closes a pointing device that was open. Once a device is closed, the pointer disappears from the screen, and you can no longer use the device for selecting enabled graphical objects. For example:

```
response to ...
  close mouse
```

If an alternate device is still open when EASEL OS/2 EE encounters an **exit** statement, the device is implicitly closed. If the device is open when EASEL OS/2 EE encounters a **change to program** statement, the device remains open for the new EASEL OS/2 EE program.

15.3.1 Controlling Input from a Pointing Device

Use the **make point disposition** to control EASEL OS/2 EE's processing of responses to input from a pointing device. This statement affects input from either pointing device: mouse or touchscreen.

Model

```
make point disposition [ mouse | touchscreen ] { normal | drop }
```

You can select either of the following dispositions:

Disposition	Meaning
normal	When EASEL OS/2 EE receives input from the pointing device, it responds to the input.

drop

When EASEL OS/2 EE receives input, it ignores the input. No input is taken until the disposition is changed to **normal**.

The **make point disposition** statement helps to handle communication with an application. When you are processing a brief transaction with an application and want to ensure that the transaction is not interrupted by input from a pointing device, specify **make point disposition drop**. After the transaction is complete, specify **make point disposition normal**.

15.4 Handling Characters Received from the Keyboard

EASEL OS/2 EE receives input from the keyboard just as if the characters were being received from a local or remote application. Thus, the following statements work with the keyboard in exactly the same way as they do with an application:

response to line	Response statement for a line of input.
response to char	Response statement for any character(s).
make string disposition	Action statement to hold, drop, or immediately handle character input.
input	Built-in function containing the received characters.

15.4.1 Matching Special Characters from the Keyboard

In a match clause in a **response to char** or **response to line** response statement, EASEL OS/2 EE can match the special characters and key combinations shown in the table below as input from the keyboard.

15.4.1.1 Keyboard Mapping

You must specify the characters shown in this table in uppercase and lowercase exactly as shown.

Key or Key Combination	Character(s) Generated
Backspace	"\b" (or "\^H")
Enter	"\n" (or "\^J")
Esc	"\e\e" (or "\^[^[")
Ctrl + @	"\^C@"
Ctrl + A	"\^A"
Ctrl + B	"\^F\^B"
Ctrl + C	"\^F\^C"
Ctrl + D	"\^F\^D"
Ctrl + E	"\^F\^E"
Ctrl + F	"\^F\^F"
Ctrl + G	"\^G"
Ctrl + H	"\^H" (or "\b")
Ctrl + I	"\^I" (or "\t") (see 15.4.1.3, Exceptions)
Ctrl + J	"\^J" (or "\n")
Ctrl + K	"\^K" (or "\v")
Ctrl + L	"\^L" (or "\f")
Ctrl + M	"\^M" (or "\r")

Key or Key Combination	Character(s) Generated
Ctrl+N	"\^N"
Ctrl+O	"\^O"
Ctrl+P	"\^P"
Ctrl+Q	"\^Q"
Ctrl+R	"\^R"
Ctrl+S	"\^S"
Ctrl+T	"\^T"
Ctrl+U	"\^U"
Ctrl+V	"\^V"
Ctrl+W	"\^W"
Ctrl+X	"\^X"
Ctrl+Y	"\^Y"
Ctrl+Z	"\^Z"
Ctrl+[	"\^[\" (or "\e\e")
Ctrl+\	"\^\"
Ctrl+]	"\^]"
Ctrl+^	"\^^"
Ctrl+_	"\^_"
Tab	"\^I" (or "\t") (see 15.4.1.3, Exceptions)
BackTab	"\^BT"
Alt+A	"\^DA"
Alt+B	"\^DB"
Alt+C	"\^DC"
Alt+D	"\^DD"
Alt+E	"\^DE"
Alt+F	"\^DF"
Alt+G	"\^DG"
Alt+H	"\^DH"
Alt+I	"\^DI"
Alt+J	"\^DJ"
Alt+K	"\^DK"
Alt+L	"\^DL"
Alt+M	"\^DM"
Alt+N	"\^DN"
Alt+O	"\^DO"
Alt+P	"\^DP"
Alt+Q	"\^DQ"
Alt+R	"\^DR"
Alt+S	"\^DS"
Alt+T	"\^DT"
Alt+U	"\^DU"
Alt+V	"\^DV"
Alt+W	"\^DW"
Alt+X	"\^DX"
Alt+Y	"\^DY"
Alt+Z	"\^DZ"
F1	"\^F1" (see 15.4.1.3, Exceptions)

Key or Key Combination	Character(s) Generated
F2	"\^F2"
F3	"\^F3"
F4	"\^F4"
F5	"\^F5"
F6	"\^F6"
F7	"\^F7"
F8	"\^F8"
F9	"\^F9"
F10	"\^F0" (see 15.4.1.3, Exceptions)
F11	"\^F_"
F12	"\^F+ "
Shift + F1	"\^B1"
Shift + F2	"\^B2"
Shift + F3	"\^B3"
Shift + F4	"\^B4"
Shift + F5	"\^B5"
Shift + F6	"\^B6"
Shift + F7	"\^B7"
Shift + F8	"\^B8"
Shift + F9	"\^B9"
Shift + F10	"\^B0"
Shift + F11	"\^B_"
Shift + F12	"\^B+ "
Ctrl + F1	"\^C1"
Ctrl + F2	"\^C2"
Ctrl + F3	"\^C3"
Ctrl + F4	"\^C4"
Ctrl + F5	"\^C5"
Ctrl + F6	"\^C6"
Ctrl + F7	"\^C7"
Ctrl + F8	"\^C8"
Ctrl + F9	"\^C9"
Ctrl + F10	"\^C0"
Ctrl + F11	"\^C_"
Ctrl + F12	"\^C+ "
Alt + F1	"\^D1"
Alt + F2	"\^D2"
Alt + F3	"\^D3"
Alt + F4	"\^D4" (see 15.4.1.3, Exceptions)
Alt + F5	"\^D5" (see 15.4.1.3, Exceptions)
Alt + F6	"\^D6" (see 15.4.1.3, Exceptions)
Alt + F7	"\^D7" (see 15.4.1.3, Exceptions)
Alt + F8	"\^D8" (see 15.4.1.3, Exceptions)
Alt + F9	"\^D9" (see 15.4.1.3, Exceptions)
Alt + F10	"\^D0" (see 15.4.1.3, Exceptions)
Alt + F11	"\^D_"
Alt + F12	"\^D+ "

Key or Key Combination	Character(s) Generated
Home	"\eH"
Up Arrow	"\eU"
Down Arrow	"\eD"
Left Arrow	"\eL"
Right Arrow	"\eR"
Page Up	"\^FT"
Page Down	"\^FB"
End	"\^FE"
Insert	"\^FI"
Delete	"\^FX"
Ctrl + Up Arrow	"\^CU"
Ctrl + Down Arrow	"\^CD"
Ctrl + Left Arrow	"\^CL"
Ctrl + Right Arrow	"\^CR"
Ctrl + Page Up	"\^CT"
Ctrl + Page Down	"\^CB"
Ctrl + Home	"\^CH"
Ctrl + End	"\^CE"
Ctrl + Print Screen	"\^C*"
Alt + 1	"\^D!"
Alt + 2	"\^D@"
Alt + 3	"\^D"
Alt + 4	"\^D\$"
Alt + 5	"\^D% "
Alt + 6	"\^D^"
Alt + 7	"\^D& "
Alt + 8	"\^D*"
Alt + 9	"\^D("
Alt + 0	"\^D) "
Alt + -	"\^D- "
Alt + =	"\^D= "

15.4.1.2 Remarks

For two-character sequences, you must respond to each character separately.

15.4.1.3 Exceptions

For CTRL-I and Tab, if you are compiling with the configuration option **KBDTABWID**=0, located in the CONFIG.ESL file, the character generated is as shown in the table. Otherwise, the characters generated are the number of spaces required to bring the current column position to the next tab stop, where the width of the tab stop is set in **KBDTABWID**.

For windowed applications, the following keys are reserved by PM and are unavailable for use in responses: F1, F10, Alt-F4 through Alt-F10.

```
# Respond to ALT-letter combinations
# First ALT-A, then ALT-B, is typed
  response to char "\^D" from keyboard
    begin guarded

      response to char "A" from keyboard
    ::
      response to char "B" from keyboard
    ::
    end
```

15.5 Providing Feedback to Selections

15.5.1 Providing Audio Feedback to Selections

EASEL OS/2 EE makes a “beep” sound when the user selects an enabled object; it makes a “boop” sound when the user misses all objects, selects a disabled object, or selects an enabled object for which there is no active response. It makes a third sound (approximately between a “beep” and a “boop”) when the user tries to select a disabled PM object.

You can control the sound frequency (in Hertz) of the “beep” and the “boop” by setting values for the following commands in the CONFIG.ESL file. The higher the value, the higher the frequency.

BEEPFREQ Sound frequency for correct selections (“beep”). Initially, this command is not in the CONFIG.ESL file; the default value of **1250** is used.

BOOPFREQ Sound frequency for incorrect selections (“boop”). Initially, this command is not in the CONFIG.ESL file; the default value of **250** is used.

You cannot change the frequency of the PM sound.

Note: In a windowed program you should probably disable EASEL OS/2 EE’s beeps and boops, to conform to PM. Specify the **-nb** option either in the EASEL.CMD file or on the command line, to disable the beeps and boops.

15.5.2 Providing Video Feedback to Selections

When EASEL OS/2 EE is being used in a noisy environment, you may want to provide visual confirmation for correct and incorrect selections.

15.5.2.1 Highlighting Correct Selections

The **highlight** action statement turns highlighting on or off. When highlighting is on, a correct selection on an object causes a frame to be displayed around the object for a moment.

Model

```
highlight { [color] COLOR | off }
```

Highlighting is turned on by specifying a color for the frame. The **highlight** statement is typically specified in a **response to start** response definition. For example:

```
response to start  
  highlight white
```

When used with the EASEL OS/2 EE Standard Color Set (listed in 5.3.3.1, "Standard EASEL OS/2 EE Color Set" on page 5-9), the frame is always drawn in *complement mode*—the inverse of the background color—to ensure that the frame is always visible. The specified color of the frame is displayed in that color when the frame is against a black background, but it changes color as necessary when it is moved to another color background.

White is the most useful color for the highlight: when inverted, it is always more visible than any other specified color. For example, it is white against a black background; black against white; green against magenta; and so on. Note that a black frame is not useful, because inverted black is not visible against any background color.

You can change the amount of time the highlight appears by setting the value of the **HLDELAY** configuration option (for "HighLight DELAY") in the CONFIG.ESL file. Initially, the value of **HLDELAY** is 8000; you can specify a value between 0 and 32000. The larger the value, the longer EASEL OS/2 EE flashes the highlight. The default value is 0; thus, if you specify HLDELAY without a value, the highlight is drawn and then immediately erased.

To turn off visual highlighting of correct selections, specify:

```
highlight off
```

Index

-g command line option 11-15

A

absolute position
 in coordinate system 5-35
accelerator keys 10-14
accel.inc file A-3
accel.inc include file 10-15
action bar
 adding at runtime 10-19
 changing region 10-19
 deleting from region 10-20
 example template 10-17
 manipulating 10-19
action bar action
 model of 10-10
action bar attribute 3-16, 10-6
action bar items
 item classes 10-21
 manipulating 10-20
 naming 10-20
action bar keyword 3-17
action bar responses 10-27
action bars
 adding items to 10-24
 built-in functions 10-28
 creating 10-9
action is statement 11-5, 11-6
action routines 11-4
 characteristics of 11-5
action statement 11-17
 tracing 24-5
action statements 6-2
 in program block 6-12
 overview 6-23
 tracing 24-3
 using 3-7
 viewports 7-47
 windows 7-47
action subroutine
 definition 3-7

add action bar command
 model 10-19
add statement 6-24, 7-4, 7-10, 8-27
add to class statement 6-37, 8-27
add to statement 6-24, 6-25, 7-10, 8-27
adding items to action bars 10-24
adding items to pull-downs 10-25
aligning
 text 7-45
ancestry
 at position attribute 5-20
 attribute 5-12
 change statement 6-27
 changing 6-33
 coordinate system 5-20
 deleting children 5-13
 in clear statement 6-26
 in desktop 3-19
 in name conversions 4-55
 in object references 5-13
 in string conversions 4-55
 members of function 6-42
 screen 5-21
 textual region 8-2
 valid parents 5-13
 visibility 5-8
ancestry function 4-42, 6-7
 with object function 6-39
ancestry of function 4-43, 6-31, 7-59, 8-42
-apio 23-9, 24-5
-apio command
 tracing 24-5
APPC support
APPC support in EASEL OS/2 EE
 ASCII to EBCDIC
 conversion 19-22
 CM considerations 19-100
 conversation information,
 getting 19-22
 DLL messages 19-4, 19-8

APPC support in EASEL OS/2 EE
(continued)

- EBCDIC to ASCII
 - conversion 19-22
- events, responding to 19-4, 19-8
- overview 19-2
- profile subroutines 19-13
- profiles, description 19-12
- reserved constants 19-98
- subroutine list 19-11

APPCAcceptStart 19-26

APPCConvertA2E() 19-28

APPCConvertE2A() 19-31

APPCEnd() 19-34

APPCFlush() 19-36

APPCGetAcceptProfileNames() 19-41

APPCGetAcceptProfile() 19-38

APPCGetDLLData() 19-43

APPCGetInfo() 19-46

APPCGetInitProfileNames() 19-53

APPCGetInitProfile() 19-50

APPCGetString() 19-55

APPCGetSystemError() 19-57

APPCReadyToReceive() 19-61

APPCReceiveNotify() 19-63

APPCRequestConfirm() 19-65

APPCRequestToSend() 19-67

APPCSendConfirmed() 19-69

APPCSendDLLData() 19-71

APPCSendError() 19-75

APPCSendString() 19-77

APPCSetAcceptProfile() 19-81

APPCSetInitProfile() 19-84

APPCStart() 19-88

APPCTestReqToSend() 19-91

append statement 4-21

application

- as global entity 14-2
- communications for 14-5
- creating 3-5
- distributing 2-8
- escape characters 14-7
- event-driven 2-5
- local 14-2
- match clause for 14-9
- overview 14-1

application (continued)

- remote 14-2
- remote, characteristics 14-5, 14-21
- responding to 14-2
- starting 14-3
- starting and stopping 14-3
- starting local 14-4
- starting, remote 14-5
- tracing communication 24-5
- writing 2-5

application Environment

Declaration 14-3, 14-7

application statement 14-2

applications

- as global entity 11-14
- match clause for 14-8
- sending escape sequences 4-15
- windowed vs fullscreen 3-19

application, event-driven 2-5

arc

- drawing 7-24, 7-26
- moving in 7-28
- specified degrees 7-24
- specified endpoint 7-26

array

- defining 4-37
- dimensions 4-37
- elements 4-38
- global 4-39
- global string variable 11-33
- initializing 4-37, 4-38
- memory 4-40
- referencing elements 4-39
- resizing 4-37, 4-40
- subscript 4-37
- value 4-38

ASCII

- file, coloring text in 8-5, 8-58
- file, default colors in 8-59
- file, inserting 8-20
- text file, reading 8-33

ASCII files

- printing 22-5

ASCII to EBCDIC conversion in APPC 19-22

- asc1115 keyword 5-46
- asc1521 keyword 5-46
- asc37 keyword 5-46
- asc510 keyword 5-46
- asc59 keyword 5-46
- asc713 keyword 5-46
- asc79 keyword 5-46
- asc814 keyword 5-46
- asc88 keyword 5-46
- at default position attribute 7-4
- at position attribute 6-26, 7-3
- attribute
 - action bar 10-6
 - border 10-4
 - maximize 10-5
 - minimize 10-4
 - no scale 10-7
 - pointer 10-7
 - primary 3-19
 - priority 7-73
 - scroll 10-6
 - system menu 10-5
 - title bar 10-4
 - visibility 7-73
- attributes 5-3
 - ancestry 5-12
 - color 5-8
 - defining 3-17
 - frame related 3-17
 - frame style 3-16
 - parameter 5-15
 - priority 5-14
 - selectability 5-5
 - visibility 5-8
- AUDClose() 18-9
- AUDErase() 18-10
- audio files
 - creating 18-21
 - deleting 18-10
 - description 18-2
 - errors 18-5, 18-13, 18-14, 18-21
 - size considerations 18-3, 18-22
 - specifying path of 18-11
- audio support
 - all, list of 18-6
 - AUDClose() 18-9
 - audio support (*continued*)
 - AUDErase() 18-10
 - AUDOpen() 18-11
 - AUDPause() 18-12
 - AUDPlay() 18-13
 - AUDQueryBusy() 18-17
 - AUDQueryElapsedTime() 18-19
 - AUDQueryPlayingTime() 18-20
 - AUDRecord() 18-21
 - AUDResume() 18-25
 - AUDSetBalance() 18-26
 - AUDSetVolume() 18-28
 - AUDStop() 18-29
 - audio support in EASEL OS/2 EE
 - audio files description 18-2
 - audio utility 18-3
 - DLL messages 18-4
 - events, responding to 18-4
 - IBM Audio Capture and Playback Adapter/A 18-2, 18-13, 18-14, 18-22, 18-23
 - integer constants 18-5, 18-7
 - overview 18-2
 - reserved constants 18-7, 18-33
 - sample program 18-31
 - sound output 18-2
 - sound source 18-2, 18-22
 - speakers, external 18-2, 18-13, 18-14, 18-26, 18-28
 - speaker, workstation 18-2, 18-13, 18-14, 18-26, 18-28
 - subroutine list 18-6
 - audio utility program 18-3
 - AUDIO.EBI 18-3
 - AUDOpen() 18-11
 - AUDPause() 18-12
 - AUDPlay() 18-13
 - AUDQueryBusy() 18-17
 - AUDQueryElapsedTime() 18-19
 - AUDQueryPlayingTime() 18-20
 - AUDRecord() 18-21
 - AUDResume() 18-25
 - AUDSetBalance() 18-26
 - AUDSetVolume() 18-28
 - AUDStop() 18-29

- automatic interface, APPC
 - description 19-14
 - overview 19-3
 - sample program 19-93
 - subroutines 19-15
- A3270.EXE application 20-2
- A5250.EXE application 21-2

B

- b 23-3
- background at function 4-43, 6-30, 8-42, 8-47
- background attribute 7-4
- background color 5-10
 - textual region 6-30
- background of function 4-43, 6-30, 7-59, 8-42
- backspace character 8-16, 8-34
 - overwriting 8-22
- BEEPFREQ 15-14
- begin statement 6-11
- BGraph module
 - GAutoScaleX 16-20
 - GAutoScaleY 16-20
 - GAutoXTitle 16-19
 - GAutoYTitle 16-19
 - GAxisTitleFont 16-18
 - GBottomX 16-20
 - GBottomY 16-20
 - GColor 16-28
 - GDataElements 16-8
 - GDataSets 16-8
 - GDataX 16-6
 - GDataY 16-6
 - GDecimalPoint 16-8
 - GFootnote 16-27
 - GFootnoteColor 16-28
 - GFootnoteFont 16-27
 - GLabels 16-22
 - GLegend 16-25
 - GLegendColor 16-26
 - GLegendFont 16-26
 - GLegendOn 16-25
 - GLineWidth 16-33
 - GMarkerColor 16-32
- BGraph module (*continued*)
 - GMarkerLines 16-31
 - GMarkerLineWidth 16-33
 - GMaxDataElements 16-23
 - GMissingDataValue 16-7
 - GMovables 16-35
 - GPatterns 16-34
 - GPieDataSet 16-12
 - GPieLabels 16-21
 - GraphOnLast 16-35
 - GraphOverlap 16-10
 - GraphPie 16-11
 - Graph3D 16-10
 - GRedNegativeBars 16-29
 - GRegion 16-3
 - GridX 16-30
 - GridY 16-30
 - GStepLineSet 16-14
 - GTickFont 16-23
 - GTitle 16-16
 - GTitleColor 16-17
 - GTopX 16-20
 - GTopY 16-20
 - GXMarkers 16-31
 - GXTickIncrement 16-24
 - GxTitle 16-17
 - GxTitleColor 16-19
 - GYMarkers 16-31
 - GYTickIncrement 16-24
 - GyTitleColor 16-19
 - G3DOutline 16-11
 - XNumTicks 16-23
 - XTicks 16-23
 - YNumTicks 16-23
 - YTicks 16-23
- binddb command 17-6
- bindfile command 17-12
- blank 14-12
- blank in match clause 14-9
- blanks
 - in program 4-2
- block definition
 - column 8-12
 - line 8-12
 - thru 8-12

- block See Block (Text)
 - text 8-12
- block statement 11-17
- block (Program) 14-14
 - active 6-10
 - active, in nested 6-15
 - definition 6-11
 - entering, responding to 6-20
 - exiting 6-13, 6-14, 6-19
 - exiting, responding to 6-16
 - guarded 6-12, 6-14, 6-18
 - guarded, exiting 6-19
 - identifier 6-11
 - inactive 6-10
 - loop in 11-26
 - nested 6-10, 6-12, 6-15
 - overview 6-10
 - response to char statement 6-21
 - response to start statement 6-20
 - resumable 6-12
- block (Text)
 - blanks 8-25
 - copying contents of 8-43, 8-45
 - emphasizing 8-30
 - in color header 8-60
 - in colored textual regions 8-55
 - overwriting 8-23, 8-25
 - removing 8-28
- boolean
 - value 11-18
- BOOPFREQ 15-14
- border
 - in shape 7-29
- border attribute 7-4, 10-4
 - size or standard 3-16
- border of function 4-43, 6-30, 7-60, 8-42
- border statement 7-29
- bottom of function 4-43, 7-55, 8-41
- boundary
 - in circle 7-21
 - in ellipse 7-21
 - in shape 7-20
 - specification 7-20
- box
 - color of 7-14
 - colored 7-14
 - hollow 7-14
 - solid 7-14
 - statement 7-13
- box statement 7-9
 - color keyword 7-13
 - solid keyword 7-13
- buffer, examining
 - 3270 20-14
 - 5250 21-14
- built-in functions 4-42, 7-54
 - action inquiry 4-44
 - in item inquiries 10-28
 - in item response 10-29
 - item inquiry 4-44
 - object attributes 7-59
 - object inquiry 4-43
 - response inquiry 4-42
 - special inquiry 4-45
 - textual region 8-41
- Business Graphics
 - action routines 16-5
 - axis boundaries 16-20
 - Axis Titles 16-17
 - colors 16-28
 - creating 16-3
 - customizing 16-15
 - footnotes 16-27
 - graph titles 16-16
 - illustrations 16-41
 - introduction 16-2
 - legends 16-25
 - line graph 16-41
 - overlapping bar graph 16-10, 16-45
 - pie graph 16-11, 16-43
 - pie wedge labels 16-21
 - public file 16-51
 - required procedures 16-3
 - rules for 16-10
 - sample program 16-38
 - scatter graph 16-44
 - stacked bar graph 16-46
 - stacked line graph 16-42

Business Graphics (*continued*)

- standard bar graph 16-44
- standard line graph 16-41
- stepped line graph 16-14, 16-42
- superimposing two graphs 16-35
- surface line graph 16-43
- 3-D bar graph 16-10
- 3-D overlapping bar graph
- three dimension ov 16-46
- 3-D stacked bar graph 16-47
- three dimensional bar graph 16-10
- three dimension o 16-11
- three dimension ov 16-46
- three-dimension 16-45
- tick mark labels 16-21
- variables 16-10
- variables, summary of 16-48
- buttons
 - defining 10-13
- by keyword 5-35

C

- C routines 25-3
- cache
 - in image regions 7-65
 - no cache 7-66
 - work area 7-66
- cache image 7-65
- cache viewport 7-66
- Canadian-French code page 4-4
- caps814 keyword 5-46
- centering
 - centering 7-45
- change graphics statement 5-12, 6-24, 6-27, 6-29, 7-10, 8-27
- change item statement
 - model 10-23
- change position statement 5-28, 5-29, 6-31, 7-50, 8-27
- change priority statement 5-15, 6-31
- change size statement 5-28, 5-30, 7-48
- change statement 6-24, 6-27, 6-29, 7-10, 8-27
- change text action statement 3-18
- change text to action
 - model of 10-8
- change to program
 - transfer control 11-11
- change to program command 3-21
- change to program statement 4-39, 6-3, 11-2, 11-10, 11-11, 15-2, 15-7
 - with global arrays in programs 11-15
 - with global entities 11-15
 - with global variables 11-33
- change to statement 7-4
- change window position 7-52
- change window position statement 5-25
- change window size
 - statement 5-25
- changes
 - enhancements 3-3
 - features, new 3-2
 - summary 3-2
- changing
 - ancestry 6-31, 6-33
 - attributes 6-28
 - border 6-30
 - color 6-29
 - parameter 6-31
 - position 6-31
 - priority 6-31
 - selectability 6-28
 - size 6-31
 - visibility 6-28
- changing position
 - window 7-52
- changing size
 - window. 7-51
- char
 - in background at statement 8-47
 - in foreground at statement 8-47
- character cell 5-43
 - intercell gap 5-43
- character set 4-2

- check boxes 9-20
- check keyword 10-22
- checked in group function 4-43
- choice definition
 - model of 10-12
- circle
 - color of 7-16
 - colored 7-16
 - solid 7-16
 - solid or hollow 7-16
 - statement 7-15
- circle statement 7-9, 7-20
 - radius keyword 7-15
- class
 - actions for 6-36
 - adding object to 6-37
 - ancestry 6-42
 - ancestry function 6-40
 - changing membership 6-38
 - defining 6-34
 - definition 3-7
 - deleting object from 6-38
 - description 6-34
 - in action statement 6-23
 - members of function 6-41
 - parameter function 6-40
 - responding to 6-35
- clear graphics statement 5-12, 6-24, 6-26, 8-27
- clear statement 6-24, 6-26, 8-27
- clipboard function 4-45
- close statement 15-7
- close subroutine 13-31
 - error messages 13-35
- code page support 4-4
- codepage
 - installation 1-4
- col in match clause 14-9, 14-10
- color
 - additional set 5-9
 - background 5-10
 - border 5-10
 - changing 6-29
 - controlling 3-20
 - ellipse 7-17
 - foreground 5-10
- color (*continued*)
 - graphical regions 7-7
 - in drawing statements 5-11
 - in regions 5-10
 - lines 7-12
 - of a box 7-14
 - of a line 7-12
 - of a polygon 7-15
 - of a shape 7-29
 - of background 8-47
 - of circle 7-16
 - of foreground 8-47
 - of pattern 7-35
 - of text 7-46, 8-5
 - of wedge 7-19
 - polygon 7-15
 - standard set 5-9
 - text segments 8-23
 - values 5-9
- color keyword 5-9, 5-10
- color library
 - EslQueryEgaColor function 13-4
 - EslSetEgaColor function 13-4
- color numbers 5-9
- color values vii
- color-attributed file 8-6
 - reading into a textual region 8-34
 - writing to 8-32
- colored textual region
 - graphics boards for 8-54
- colored textual region statement 8-54
- colored textual regions
 - color 8-54
- coloring
 - text 8-55
- column
 - in background at statement 8-47
 - in foreground at statement 8-47
- column size of function 4-43, 8-46
- combination boxes 9-15
- command line options
 - apio 23-9
 - debug 23-10
 - e 23-10

command line options (*continued*)

- f 23-10
- g 23-10
- m 23-4, 23-10
- nb 23-10
- terp 23-10
- apio
- b 23-3
- d
- debug 23-3
- e 23-3
- f 23-4
- fullscreen 23-4
- g 23-4
- lx 23-4
- m
- nb
- SQL 17-30
- terp
- comment
 - in color header 8-60, 8-61
- comments
 - in ASCII text file 8-60
 - in program 4-3
- communicating with host
 - 3270 20-15
 - 5250 21-49
- communications 24-5
 - characteristics 14-5
 - with application 14-7
- communications port
 - characteristics 14-21
- compile command 3-12, 3-13
- compile-time value 4-11
- COMPILE.CMD 3-19
- COMPILE.CMD OS/2 Command File 23-3
- compiling 3-12
- complement mode 15-15
- components
 - of an object 5-3
- conditional statements
 - Nested 11-20
- configuration commands 17-7
- configuration options
 - BEEPFREQ 23-5

configuration options (*continued*)

- BOOPFREQ 23-5
- CODEPAGE 23-6
- ESLERR 23-6
- EXETYPE 23-6
- FontCNT 23-6
- FontPATH 23-6
- GLBMAX 23-6
- GLBSIZE 23-6
- HLDELAY 23-6
- INCPATH 23-6
- INCTYPE 23-6
- KBDTSBWID 23-7
- LOCALDLL 23-7
- MODPATH 23-7
- POINTER 23-7
- PRGMAX 23-7
- PRGSize 23-7
- REMOTE 23-7
- REMOTEDLL 23-7
- SLEEPMAX 23-7
- TABWIDTH 23-7
- TEXTPATH 23-7
- TEXTTYPE 23-8
- XMAX 23-8
- YMAX 23-8
- configuring CM for EASEL OS/2 EE APPC 19-100
- CONFIG.ESL 1-4
- CONFIG.ESL file 3-13, 5-42, 11-8, 23-5
 - editing during install 3-13
- CONFIG.SYS 1-4
- CONFIG.SYS file 15-14, 15-15
 - CODEPAGE statement 4-4
 - COUNTRY statement 4-4
 - DEVINFO statement 4-4
- confirmation processing, APPC
 - description 19-16
 - overview 19-5
 - subroutines 19-17
- Connect 20-19
- connecting
 - 3270 20-14
 - 5250 21-14

- constant 4-16
- constant statement 4-16
- constant statement(s) 3-6
- controlled interface, APPC
 - description 19-18
 - overview 19-4
 - sample program 19-95
 - subroutines 19-21
- controlling program
 - Timing 11-29
- conversion
 - to uppercase 5-46
- conversions
 - boolean/string 4-54
 - integer/floating point 4-54
 - string/floating point 4-53
 - string/integer 4-53
 - string/name 4-55
- coordinate
 - of selection 6-9
- coordinate system
 - ancestor 5-20
 - graphical object 5-18
 - object 5-18
 - of regions 5-22
 - of screen 5-16
 - of viewport 5-22
 - of window 5-22
 - textual region 5-20
- coordinates
 - of a pattern 7-34
- copy statement 4-20
- courier font 5-47
- creating
 - action bars 10-9
- CText module 8-58, 8-60
- CTR_File variable 8-58, 8-61
- CTR_Header variable 8-58, 8-61
- CTR_Read action subroutine 8-58
- CTR_Region variable 8-58, 8-61
- cursor
 - graphics 5-33
 - in drawing statement 5-32
 - in objects 5-32
 - mouse 15-2
 - text 5-35

- cursor movement
 - in box statement 5-33
 - in circle statement 5-33
 - in draw statement 5-33
 - in move statement 5-33
- cursors
 - graphics 7-10
 - moving 7-10

D

- databits 14-22
- date
 - library functions 13-7
- date function 4-45
- date library
 - DaySpan function 13-7
 - LocalDate function 13-8
 - LocalTime function 13-8
 - Validate function 13-9
 - WeekDay function 13-10
- date library functions 13-7
- datelib.inc file A-4
- debug 23-3
- debugging 24-1
- Declaring a stimulus 14-29
- deemphasize statement 8-30
- default
 - in color header 8-59, 8-61
- DefineWatch 20-35
- defining template
 - action bar 10-10
- delete action bar command
 - model 10-20
- delete from class statement 6-38, 8-27
- delete item statement
 - model 10-22, 10-26
- delete statement 6-24, 6-28, 8-27
- deleting from classes 6-26, 6-27
- deleting items from regions 10-26
- desktop
 - ancestor 5-12, 5-21
 - screen size 5-18
- desktop keyword 3-19

- development process 2-7
- device units 3-16
- DEVINFO= KBD statement 4-6
- dialog box
 - sample code 9-24
- dialog box definition
 - model 9-2
- dialog boxes 9-2
 - sample program 9-24
- dialog control objects 9-7
- Dialog region 9-5
- dialog units 3-16
- digitized audio files 18-2
- direct commands 17-3
- directory
 - created 1-7
- disable statement 5-7, 6-28
- disabled keyword 5-5
- Disconnect 20-20
- disconnecting
 - 3270 20-14
 - 5250 21-14
- display 5-16
- display command 17-22
- display resolution B-5
- displaying
 - text 7-44
- DLL declaration, audio 18-4
- DLL declaration, in APPC 19-5
- DLL messages, APPC 19-4, 19-8
- DLL messages, audio 18-4
- draw arc statement 7-19, 7-20, 7-24
 - center 7-24
 - clockwise keyword 7-25, 7-26
 - counter clockwise
 - keyword 7-25, 7-26
- draw statement 5-19, 7-9, 7-11, 7-14, 7-20
 - color keyword 7-11
- drawing
 - line 7-11
- drawing statement
 - description 5-4
 - for text 7-44
 - graphical objects 7-9
 - graphics cursor 5-33
- drawing statement (*continued*)
 - placement 7-9
 - set of 7-30
 - text cursor 5-35, 5-37
- drawing statements
 - programs 8-7
 - textual region 8-2, 8-7
 - where to place 8-7
- drop 14-16
- drop keyword 15-7
- drop-down combination box 9-15
- dynamic commands 17-26

E

- e 23-3
- EASEL OS/2 EE
 - application, creating 3-5
 - benefits 2-3
 - compiles 3-15
 - compiling 3-12
 - development process 2-7
 - executes 3-15
 - executing 3-12
 - fonts 5-45
 - front-end application 2-5
 - installing 1-4
 - introduction 2-2
 - library functions 13-4
 - program construction 3-6
 - programming 3-1
 - sample program 3-9
 - screen 5-16
- EASEL OS/2 EE library
 - functions 13-4
- EASEL OS/2 EE program
 - exiting 11-9
- EASEL OS/2 EE regions
 - plotting 22-2
 - printing 22-2
- EASEL OS/2 EE sample programs
 - textual regions 8-49
 - using CText module 8-62
- EASEL.CMD file 24-5
- EASEL.CMD OS/2 Command
 - File 23-9

- EBCDIC to ASCII conversion in
 - APPC 19-22
- EGA
 - fullscreen colors 5-9
- egacolor.inc
 - include file 13-4
- egacolor.inc include file 13-4, A-5
- EGA/VGA
 - color values 13-5
 - ENTRY color values 13-5
- EGA/VGA color values 13-5
- elements
 - referencing in array 4-39
- ellipse
 - color of 7-17
 - colored 7-17
 - solid 7-17
 - solid or hollow 7-17
 - statement 7-16
- ellipse statement 7-9, 7-16, 7-20
- emphasize statement 8-29
- emphasizing
 - in text 8-29
 - segment 8-29
- EmulateAndWatch 20-53
- Emulate3270 20-52
- enable statement 5-7, 6-28
- enabled 6-4, 6-6
- enabled keyword 3-9, 5-5
- end loop statement 11-21
- end shape statement 7-20
- end statement 6-11
- EnterString 20-49
- entity
 - local 11-14
- entry field definition
 - model 9-8
- entry fields
 - multiple-line 9-9
 - single-line 9-8
- entry groups of dialog controls 9-7
- environment declaration
 - application 3-6
 - font 3-6
 - module 3-6
- environment files 23-1
- environmental definition 14-2
- eparse command 3-19
- error command 17-9
- error handling 21-60
 - SQL 17-27
 - 3270 20-64
 - 5250 21-60
- error messages C-1
 - fatal runtime C-1
 - for close subroutine 13-35
 - input subroutines 13-40
 - open subroutine 13-34
 - output subroutines 13-43
 - parameter adjustment subrou-
tines 13-46
 - plotting 22-4
 - printing 22-7
 - SQL 17-32
 - 3270 20-66
 - 5250 21-62
- error messages for math subrou-
tines 13-13
- error status variables
 - 3270 20-64
 - 5250 21-60
- errorlevel function 4-45
- errorlog 24-2
 - sending values to 24-3
 - text in 24-3, 24-4, 24-5
- errorlog file
 - using 24-2
- errorlog keyword
 - in send statement 24-4
- escape sequences
 - in files to be read in 8-34
 - in string literals 4-14
 - inserted strings 8-16
 - overwriting 8-22
 - quotation mark 4-15
 - special keys 15-9
- eslappc.inc include files A-6
- eslaudio.inc include files A-12
- esllib.inc include files A-14
- ESLPR.EXE utility
 - header keyword 22-6

ESLPR.EXE utility (*continued*)

- lpt1 keyword 22-6
- lpt2 keyword 22-6
- lpt3 keyword 22-6
- pagesize keyword 22-6
- ESL1002 C-3
- ESL1003 C-3
- ESL1004 C-3
- ESL1005 C-3
- ESL1006 C-4
- ESL1007 C-4
- ESL1008 C-4
- ESL1009 C-4
- ESL1010 C-5
- ESL1011 C-5
- ESL1012 C-5
- evenpar 14-21
- event notifications, APPC 19-4, 19-8
- event notifications, audio 18-4
- eventnumber built-in function 14-30
- eventnumber function 4-42
- eventparam built-in function 14-30
- eventparam function 4-42
- executing 3-12
- exists function 4-43, 7-60, 8-42
- exit
 - from a loop 11-24
- exit action statement 11-9
- exit command 17-20
- exit statement 11-2, 11-9, 15-7
- expression 4-47
 - compile-time 4-47
 - compound 4-47
- extract from statement 4-22, 14-14
 - column 4-32
 - float 4-30
 - last 4-26
 - marker 4-24
 - number 4-29
 - skip clause 4-22, 4-24
 - take clause 4-22, 4-24
 - text 4-31
 - with textual built-in function 8-46

extract from statement (*continued*)

- word 4-27

F

- f 23-4
- fatal runtime errors C-1
 - ESL1002 C-3
 - ESL1003 C-3
 - ESL1004 C-3
 - ESL1005 C-3
 - ESL1006 C-4
 - ESL1007 C-4
 - ESL1008 C-4
 - ESL1009 C-4
 - ESL1010 C-5
 - ESL1011 C-5
 - ESL1012 C-5
- features, new 3-2
- fields utility 20-8
 - retriving information 20-8
 - sample output 20-10, 21-11
 - 3270 20-2
 - 5250 21-3
- file contents
 - of Image regions 7-68
- file i/o library 13-26
 - subroutines 13-26
- file i/o subroutines
 - ASCII files 13-26
 - parameter types 13-27
 - record definition 13-27
- fileio.inc file A-15
- files
 - installation 1-7
- find statement
 - column keyword 8-34
 - found function 4-44
 - from keyword 8-34
 - line keyword 8-34
- FindLastNonBlankLine 20-29
- fixed-cell font 5-45
- flag
 - fullscreen 3-19
- font
 - and ancestry 5-43

- font (*continued*)
 - and screen size statement 5-45
 - attributes 5-49
 - character cells 5-43
 - character descenders 5-44
 - defining 5-48
 - EASEL OS/2 EE types 5-45
 - environment declaration 3-6
 - fixed-cell 5-44, 5-45, 5-46
 - fixed-width 5-47
 - image 5-49
 - in pattern 7-36
 - in textual region 5-50
 - inheritance 5-43
 - maximum in program 5-42
 - monospaced 5-47
 - names and sizes 5-45
 - of text 7-46
 - pixels 5-43
 - PM types 5-47
 - portability 5-44
 - proportional 5-47
 - resolution-dependent 5-44, 5-45, 5-46
 - scalable 5-50
 - uppercase 5-46
 - font attribute 5-50
 - font is statement 5-48
 - font statement 5-42
 - FontCNT option 5-42
 - fonts
 - textual region 8-3, 8-6
 - for each selected line statement
 - model 9-13
 - foreground at function 4-43, 6-30, 8-42, 8-47
 - foreground color 5-8, 5-10
 - textual region 6-30
 - foreground of function 4-43, 6-30, 7-59, 8-42
 - found function 4-44, 8-34, 8-43, 8-48
 - frame attributes 3-17
 - frame oriented syntax 3-16
 - frame related attributes 3-17
 - frames
 - defining 3-17
 - freysize function 4-45, 11-34, 11-35
 - front-end application 2-5
 - fs command 17-14
 - full-duplex conversations 19-2
 - fullscreen 23-4
 - fullscreen flag 3-19
 - fullscreen vs windowed
 - applications 3-19
 - function declaration 11-2
 - functions 11-4
 - writing 25-3
- ## G
- g 23-4
 - g command line option 11-34
 - GAutoScaleX 16-20
 - GAutoScaleY 16-20
 - GAutoXTitle 16-19
 - GAutoYTitle 16-19
 - GAxisTitleFont 16-18
 - GBottomX 16-20
 - GBottomY 16-20
 - GColor 16-28
 - GDataElements 16-8
 - GDataSets 16-8
 - GDataX 16-6
 - GDataY 16-6
 - GDecimalPoint 16-8
 - GetCursorPosition 20-25
 - GetFieldAttributes 20-24
 - GetFieldLength 20-23
 - GetFieldNumber 20-26
 - GetFieldPosition 20-27
 - GetSessionInfo 20-62
 - GetSessionStatus 20-60
 - GetXstatus 20-57
 - GFootnote 16-27
 - GFootnoteColor 16-28
 - GFootnoteFont 16-27
 - GLabels 16-22
 - GLegend 16-25
 - GLegendColor 16-26

- GLegendFont 16-26
- GLegendOn 16-25
- GLineWidth 16-33
- global entity 11-14
- global items
 - defining 10-14
- global memory 11-33
- GMarkerColor 16-32
- GMarkerLines 16-31
- GMarkerLineWidth 16-33
- GMaxDataElements 16-23
- GMissingDataValue 16-7
- GMovables 16-35
- GPatterns 16-34
- GPieDataSet 16-12
- GPieLabels 16-21
- graphical object
 - coordinate system 5-18
 - graphics cursor 5-33
- graphical objects 7-2
 - built-in functions for 7-54
 - built-in functions 7-54
 - drawing statements 7-9
- graphical region
 - ancestry 7-6, 7-7
 - and ancestry 7-6
 - attributes 7-5
 - background 7-7
 - color 7-7
 - coordinate system 5-18
 - default attributes 7-5
 - defining 7-4
 - foreground 7-8
 - graphics cursor 5-33
 - position 7-6
 - size 7-6
 - size and position of 7-6
 - statement 7-4
 - text, image 7-44
 - text, outline 7-45
 - viewport 7-47, 8-36
 - viewport, size of 7-48
 - window 7-47, 8-36
 - window, changing position 7-52
 - window, changing size of 7-51
- graphical region definition
 - border keyword 7-5
 - color keyword 7-5
 - default keyword 7-5
 - disabled keyword 7-5
 - enabled keyword 7-5
 - foreground keyword 7-5
 - invisible keyword 7-5
 - parameter keyword 7-5
 - position keyword 7-5
 - visible keyword 7-5
- graphical region statement 7-4
 - border keyword 10-3
- graphical regions
 - viewports 7-47
 - windows 7-47
- graphical region, changing position
 - of
 - viewport 7-50
- graphics boards
 - EGA 8-54
 - VGA 8-54
- graphics cursor 5-33
- GraphOnLast 16-35
- GraphOverlap 16-10
- GraphPie 16-11
- Graph3D 16-10
- GRedNegativeBars 16-29
- GRegion 16-3
- GridX 16-30
- GridY 16-30
- group box definition
 - size in 9-23
- group boxes 9-23
- GStepLineSet 16-14
- GTickFont 16-23
- GTitle 16-16
- GTitleColor 16-17
- GTitleFont 16-16
- GTopX 16-20
- GTopY 16-20
- guarded block
 - definition 6-11
 - description 6-18
- GXMarkers 16-31

- GXTickIncrement 16-24
- GxTitle 16-17
- GxTitleColor 16-19
- GYMarkers 16-31
- GYTickIncrement 16-24
- GyTitleColor 16-19
- G3DOutline 16-11

H

- half-duplex conversations 19-2
- handle of 7-61
- handle of function 4-43, 8-42
- hardcopy support 22-2
- helv font 5-47
- highlight statement 15-14
- HLDELAY 15-15
- hold 14-16
- horizontal scroll bar keyword 3-17

I

- Identifier
 - converting to string 4-55
 - description 4-8
 - predefined 4-57
- identifiers vii
- if statement 11-2
- if/then/else statement 11-17, 11-18, 11-21
- ignore tab
 - in multiline entry field
 - definition 9-10
- image font 5-49
- image region definition
 - color keyword 7-64
 - default keyword 7-64
 - file keyword 7-68
 - foreground keyword 7-64
 - model of 7-64
 - no scale specification 7-68
 - position keyword 7-64
 - preserve foreground
 - statement 7-67
- Image regions 7-63
 - background colors 7-65
- Image regions (*continued*)
 - bitmap file formats 7-63
 - cache specifications 7-65
 - defining 7-64
 - file contents of 7-68
 - foreground colors 7-65
 - operations 7-64
- in line statement
 - deselect keyword 9-14
 - select keyword 9-14
- include file 13-2
- include files A-1
 - accel.inc A-3
 - APPC, overview 19-5
 - audio, overview 18-3
 - datelib.inc A-4
 - egacolor.inc A-5
 - eslappc.h 19-43, 19-71
 - eslappc.inc 19-5, A-6
 - eslaudio.inc 18-3, 18-7, 18-9, 18-10, 18-11, 18-12, 18-13, 18-17, 18-19, 18-20, 18-21, 18-25, 18-26, 18-28, 18-29, A-12
 - esllib.inc A-14
 - fileio.inc A-15
 - mathlib.inc A-16
 - message.inc A-18
 - pmptr.inc A-20
- include statement 3-6, 11-2, 11-8
 - ASCII text file 11-8
 - include statement 11-8
 - nested files 11-8
- INCTYPE 11-8
- info command 17-11
- Init3270 20-17
- input
 - subroutines 13-35
- input function 4-22, 4-42, 6-7, 6-8, 14-14, 15-9
- input subroutines 13-35
 - error messages for 13-40
- inquiry
 - functions for 7-54
 - position 7-54
 - size 7-54

- inquiry commands 17-21
- insert statement 8-14, 8-44, 8-45
 - colored keyword 8-14
 - file keyword 8-14
 - from keyword 8-14
 - ioerror function 4-44
 - thru keyword 8-14
- inserting
 - block (Text) 8-18
 - color-attributed file 8-20
 - segment 8-17
 - string 8-15
 - text 8-14
- installation 1-1, 1-13
 - Development System 1-4
 - directory 1-7
 - diskettes 1-6
 - error messages 1-13
 - files, added 1-7
 - Runtime Facility 1-5
- installing EASEL OS/2 EE 1-4
- integer constants
 - in APPC support 19-5, 19-8, 19-10, 19-23, 19-24, 19-43, 19-57, 19-58, 19-59, 19-71, 19-98
 - in audio support 18-5, 18-7, 18-17
- interrupt response 6-19
- introduction to audio support 18-2
- introduction to EASEL OS/2 EE
 - APPC support 19-2
- invisible keyword 5-8
- invoke action statement 11-10
- ioerror function 4-44, 8-20, 8-43, 8-61
- is checked function 4-43, 4-44
- is enabled function 4-44
- item class definition
 - model 10-21
- item keyword 10-22
- item parameters 10-16
- item reference keyword
 - model of 10-14
- items
 - changing text of 10-23
 - checking and unchecking 10-22

- items (*continued*)
 - enabling and disabling 10-22

K

- KBDTABWID keyword 15-12
- key
 - attributes 7-2
 - color 7-4
 - default attributes 7-2
- key statement 7-2
 - color keyword 5-3, 7-2
 - default keyword 5-3, 7-2
 - disabled keyword 7-2
 - enabled keyword 7-2
 - invisible keyword 7-2
 - parameter keyword 7-2
 - position keyword 5-3, 7-2
 - priority keyword 7-2
 - visible keyword 7-2
- keyboard 14-16
 - layout 4-6
 - supported 4-5
- keyboard mapping table 15-9
- keys
 - objects 7-2
- Keyword 4-8

L

- language
 - supported 4-4
- large keyword 5-46
- Layout.CUA
 - pulldown choice 12-25
- Layout/CUA 12-2, 12-4, 12-5, 12-7, 12-8
 - action bar 12-14
 - action bar choice 12-25
 - adding objects 12-7
 - aligning objects 12-13
 - alignment options 12-13
 - automatic dialog box
 - calling 12-23
 - check box 12-31
 - clipboard 12-16

Layout/CUA (continued)

- code generation 12-21
 - combination box 12-36
 - creating interface 12-7
 - deleting objects 12-12
 - dialog box subroutines 12-21
 - dialog boxes 12-8
 - drop-down combination box 12-36
 - drop-down list 12-36
 - entry field, multiline 12-35
 - entry filed 12-33
 - group box 12-34
 - help system 12-5
 - hiding/showing windows 12-14
 - introduction 12-2
 - list box 12-32
 - moving objects 12-12
 - multiline entry field 12-35
 - object information 12-13, 12-24
 - primary window 12-4, 12-24
 - pull-down 12-14
 - pull-down choices 12-23
 - push button 12-29
 - push buttons 12-23
 - query subroutine 12-22
 - radio button 12-30
 - resizing objects 12-12
 - secondary window 12-8, 12-27
 - selection fields 12-15
 - setup subroutine 12-21
 - static box 12-34
 - undo 12-14
 - using 12-4
- lcaps keyword 5-46
- leave block statement 6-10, 6-13, 6-14, 6-19
- leave loop statement 6-19, 11-2, 11-17, 11-21, 11-24, 11-25
- left of function 4-43, 7-55, 8-41
- length of function 4-19, 4-45
- Libraries
- stimulus 14-29
 - writing 25-1, 25-3
- library
- functions 13-3

library (continued)

- subroutines 13-2
- library functions 13-3
- library subroutines 13-2
- Library Subroutines and Functions 13-1
- library, date
- functions 13-7
- library, EASEL OS/2 EE
- functions 13-4
- line
- colored 7-12
 - copying contents of 8-43
 - drawing 7-11
 - in background at statement 8-47
 - in foreground at statement 8-47
 - with textual function 8-43
- line graph 16-41
- line height of
- line size of function 4-43, 8-47
- textual region 8-47
- list box definition
- model 9-12
- list boxes 9-11
- literal 4-12
- boolean 4-15
 - floating point 4-12
 - integer 4-12
 - string 4-13
- local applications
- writing 25-1
- local entity 11-14
- loop
- block (Program) 11-26
 - exit from 11-22, 11-24
 - nested 11-25
- loop statement 11-3, 11-17, 11-21
- lx 23-4

M

- m 23-4
- m command line option 11-15
- machine requirements 1-2, B-2
- make block statement 8-5, 8-55

- make block/segment COLOR statement
 - background keyword 8-56
 - column keyword 8-56
 - foreground keyword 8-56
 - line keyword 8-56
 - thru keyword 8-56
- make border statement 6-30
- make color statement 6-29
- make cursor invisible 8-30
- make cursor statement 8-30
- make cursor visible 8-30
- make invisible statement 5-8, 6-28
- make item parameter
 - model of 10-17
- make parameter statement 6-31
- make point disposition statement
 - drop keyword 15-7
 - mouse keyword 15-7
- make segment statement 8-5, 8-55
- make statement
 - block keyword 8-56
 - disposition keyword 15-7
 - point keyword 15-7
 - segment keyword 8-56
- make string disposition 15-9
- make string disposition statement 14-16
- make visible statement 5-8, 6-28
- manipulating 10-19
- mapped conversations 19-2
- margin statement 7-9
- marker function 4-24, 4-45
- markpar 14-21
- master
 - ancestor 5-12, 5-21
 - screen size 5-18
- master keyword 3-19
- match
 - multiple 14-13
- match clause 6-21, 15-9
- math
 - subroutines 13-11
- math library
 - ABS function 13-14
 - ARCCOS function 13-14
 - math library (*continued*)
 - ARCSIN function 13-15
 - ARCTAN function 13-15
 - ARCTAN2 function 13-15
 - CEIL function 13-16
 - COS function 13-16
 - E function 13-17
 - EXP function 13-17
 - FLOOR function 13-17
 - function 13-11
 - LN function 13-18
 - LOG function 13-18
 - MAX subroutine 13-21
 - MAXINT subroutine 13-21
 - MEAN subroutine 13-21
 - MEANINT subroutine 13-21
 - MEDIAN subroutine 13-22
 - MEDIANINT subroutine 13-22
 - MIN subroutine 13-23
 - MININT subroutine 13-23
 - MOD function 13-18
 - PI function 13-19
 - POWER function 13-19
 - SIN function 13-20
 - SQRT function 13-20
 - STDDEV subroutine 13-23
 - STDDEVINT subroutine 13-23
 - SUM subroutine 13-24
 - SUMINT subroutine 13-24
 - TAN function 13-20
 - VARIANCE subroutine 13-24
 - VARIANCEINT subroutine 13-25
 - math library functions 13-11
 - math subroutines 13-11
 - error messages for 13-13
 - mathlib.inc file A-16
 - maximize attribute 3-16, 10-5
 - maximize button 3-17
 - mcaps keyword 5-46
 - medium keyword 5-46
 - medvt10 keyword 5-46
 - medvt12 keyword 5-46
 - medvt14 keyword 5-46
 - medvt16 keyword 5-46
 - medvt20 keyword 5-46

- medv24 keyword 5-46
- member function 4-45
- members of function 4-45, 6-41
- memory
 - allocating local 11-34
 - changing allocation in EASEL.BAT 11-34
 - global 11-33
 - global and local 11-14
 - global, allocating 11-34
 - global, restrictions in use 11-33
 - global, uses of 11-33
 - handling within program 11-34
 - runtime configurations 11-33
- message boxes 13-49
- message.inc file A-18
- minimize attribute 3-16, 10-4
- minimize button keyword 3-17
- miscellaneous action routines 20-16
 - 3270 20-16
 - 5250 21-16
- mixing regions 5-39
- mnemonics
 - in action bars 10-15
- Mnemonics with dialog controls 9-7
- mod operator 4-49
- mode
 - textual line statement 9-14
- model
 - check box definition 9-20
 - checked built-in function 9-20
 - checked in group 9-20
 - group box definition 9-23
 - line statement 9-14
 - push button definition 9-17
 - radio button definition 9-19
 - selected line from function 9-14
 - selected line function 9-14
 - static text definition 9-22
- module statement 8-60
 - BGraph keyword 4-57
 - CText keyword 4-57
 - M3270 keyword 4-57
 - M5250 keyword 4-57
- module statement (*continued*)
 - 3270 20-3
 - 5250 21-3
- modules 4-57
- monitor 5-16
- mouse 15-2
 - cursor 15-2
 - pointer appearance 15-2
- mouse pointer appearance
- move arc statement 7-19, 7-21
- move statement 7-9, 7-10, 7-21
- move to statement 6-26
- moving
 - graphics cursor 7-10
 - text cursor 7-10
- Multilingual code page 4-4
- multiple-line entry fields 9-9
- M3270 module 20-2
- M5250 module 21-2
- M5250 Support
 - watch facility 21-33
- M52_AutoWait variable 21-52
- M52_CheckIndicators 21-56
- M52_Connect 21-19
- M52_DefineWatch 21-39
- M52_Disconnect 21-20
- M52_EnterString 21-54
- M52_FindLastNonBlankLine 21-29
- M52_GetCursorPosition 21-25
- M52_GetFieldAttributes 21-24
- M52_GetFieldLength 21-23
- M52_GetFieldNumber 21-26
- M52_GetFieldPosition 21-27
- M52_GetSessionInfo 21-57
- M52_GiveUpTime 21-45
- M52_InformWhenIloff 21-34
- M52_Init 21-17
- M52_PressCMDkey 21-52
- M52_PressENTER 21-51
- M52_PressKey 21-49
- M52_PrintScreen 21-59
- M52_ReadColoredScreen 21-31
- M52_ReadField 21-22
- M52_ReadLine 21-28
- M52_ReadScreen 21-30

- M52_ScanScreen 21-21
- M52_Stop 21-18
- M52_TypeString 21-53
- M52_WaitForIloff 21-36
- M52_Watch 21-45
- M52_WatchAndWait 21-47
- M52_WatchCommandString 21-41
- M52_WatchForChar 21-43
- M52_WatchForCursorAt 21-40
- M52_WatchForCursorNotAt 21-41
- M52_WatchForIloff 21-42
- M52_WatchForNotChar 21-44
- M52_WriteField 21-55

N

- names command 17-23
- National Language Support
 - code page 4-4
 - date and time 4-7
 - keyboards 4-5
 - languages 4-4
- nb 23-10
- nested
 - pattern 7-30, 7-35
- nested blocks 6-10, 6-12, 6-15
- nested include files 11-8
- new-line character 8-16, 8-34
 - overwriting 8-22
- no scale attribute 10-7
- no scale keyword 3-17
- non-string data, sending/receiving
 - in APPC 19-15
- nopar 14-21
- Nordic code page 4-4
- normal 14-16
- normal keyword 15-7
- notation conventions v
- null command 17-10
- number 14-11
- number in match clause 14-9

O

- object
 - adding to class 6-34, 6-37, 6-38
 - ancestry 5-12
 - class membership 6-38
 - color 5-8
 - components of 5-3
 - coordinate system 5-18, 7-10
 - creating 6-24
 - creating identifiers 7-3
 - definition 3-7
 - deleting 6-28
 - deleting from class 6-38
 - name 5-3
 - origin 5-18
 - parameter 5-15
 - position 5-3
 - priority 5-14
 - selectability 5-5
 - siblings 5-12
 - type 5-3
 - types of 5-2
 - visibility 5-8
- object contents
 - adding 6-25
 - deleting 6-26
 - replacing 6-27
- object function 4-42, 6-7, 6-8, 8-43
 - with a class 6-39
 - with ancestry function 6-39
 - with parameter function 6-40
- objects 3-8
 - graphical 7-2
- oddpair 14-21
- open statement 15-6, 15-7
- open subroutine 13-31
 - error messages 13-34
- operands 4-47
- operating environment 1-2, B-1
- operators 4-47
 - arithmetic, precedence in 4-49
 - arithmetic, table of 4-48
 - boolean, precedence in 4-51
 - boolean, table of 4-51
 - relational, precedence in 4-51

- operators (*continued*)
 - relational, table of 4-50
- OPTION_STRING parameters 22-3
- OS/2 Command File
 - COMPILE.CMD 23-3
- output command 17-13
- output devices 22-2
- output subroutines 13-41
 - error messages for 13-43
- overlapping bar graph 16-10, 16-45
- Overview 13-2
- overview of APPC support 19-2
- overview of audio support 18-2
- overwrite statement
 - column keyword 8-20
 - line keyword 8-20
 - segment keyword 8-20
 - text cursor 8-21
- overwriting
 - string 8-21

P

- parameter attribute 5-15
- parameter function 4-22, 4-42, 5-15, 6-7, 6-31, 8-42, 11-17
 - with object function 6-40
- parameter of function 4-43, 5-15, 6-40, 7-61, 8-42
- parent
 - valid types 5-13
- parm adjustment subroutines
 - error messages for 13-46
- pattern
 - box statement 7-38
 - color 7-35
 - coordinate system 7-34
 - definition 3-7
 - draw statement 7-34
 - font in a 7-36
 - nested 7-30, 7-39
 - reference point of 7-33
 - rotate specification 7-37
 - rotating a 7-37
 - scale factor 7-36
 - scale specification 7-36
- pattern (*continued*)
 - text 7-39
- pattern is statement 7-30
- pattern statement 7-32
- patterns
 - nested 7-35
- pie charts
 - Business Graphics 7-18
- pie graph 16-11, 16-43
- pixels 5-43
- plot action statement 22-3
- plotting
 - error messages 22-4
- plotting
 - EASEL OS/2 EE regions 22-2
 - hardcopy support 22-2
 - OPTION_STRING
 - parameters 22-3
 - output devices 22-2
 - plot action statement 22-3
- PM fonts 5-47
- pmptr.inc file A-20
- pointer attribute 10-7
- POINTER command 1-4
- pointer keyword 3-17
- pointing device
 - as global entity 11-14
 - default 15-6
 - mouse 15-2
 - touchscreen 15-2
- pointing devices 3-20
 - overview 15-2
 - set pointer to statement 15-2
- polygon
 - color of 7-15
 - colored 7-15
 - statement 7-14
- polygon statement 7-9
 - definition 7-14
- portability B-5
- Portuguese code page 4-4
- position
 - inquiry 7-54
- precision statement 4-13
- PressENTER 20-46

- PressKey 20-45
- PressPFKey 20-47
- PRI file 4-58
- primary attribute 3-19
- primary keyword 3-8, 3-17
- printing
 - ASCII files 22-5
 - EASEL OS/2 EE regions 22-2
 - error messages 22-7
 - example 22-6
 - hardcopy support 22-2
 - optional parameters 22-5
 - output devices 22-2
 - utilities 22-5
- Print3270Screen 20-59
- priority attribute 5-14, 7-73
- priority of function 4-43, 6-31, 7-62, 8-42
- profiles, APPC
 - description 19-12
 - overview 19-2
 - setting/querying 19-12
 - subroutines 19-13
- program 3-6
 - compile 3-15
 - components, order 3-6
 - constructing 3-8
 - control 11-8
 - execute 3-15
 - sample 3-9
 - textual regions 8-49
- program construction 3-6
- program flow 11-2
- program flow statements 11-4, 11-8, 11-17
- program requirements 1-3, B-4
- prompt command 17-8
- PUB file 4-57
- pulldown action
 - model of 10-11
- pulldowns
 - defining 10-11
 - defining choices 10-12
- push buttons 9-17

R

- radio buttons 9-18
 - check statement 9-19
 - uncheck statement 9-19
- read statement 7-70, 8-20
 - colored keyword 8-33, 9-13
 - file keyword 8-33, 9-13
 - ioerror function 4-44, 8-33
 - model 9-13
 - text cursor 8-33
- ReadColoredScreen 20-31
- ReadField 20-22
- ReadLine 20-28
- ReadScreen 20-30
- redrawing
 - specified object 5-52
- refresh statement 5-52
- region
 - mixing types 5-39
 - multiple 5-31
 - position attribute 5-23
 - size attribute 5-23
- region actions 10-8
- Region Attributes 10-2
 - model of 10-3
- Region Attributes and Action Bars 10-1
- region keyword 3-8
- relative position
 - in coordinate system 5-35
- remove statement
 - from keyword 9-13
 - line 8-28
 - model 9-13
 - text cursor 8-28
- removing
 - segment 8-28
- required procedures
 - 3270 20-3
 - 5250 21-3
- reserved constants
 - 3270 20-74
 - 5250 21-70
- reserved constants, APPC 19-98

- reserved constants, audio 18-33
- reserved variables
 - 3270 20-71
 - 5250 21-67
- resize statement 4-40
- resolution
 - monitor 5-17
- resolution-dependent font 5-45
- responding to
 - action bar items 6-6
 - application input 6-8
 - class 6-35, 6-36
 - entering a block 6-3
 - keyboard input 6-8
 - mouse double click 6-4
 - objects 6-4
 - program startup 6-3
 - stimuli 6-17, 6-19
 - stimuli types 6-3
 - system menu close 6-4
 - window activation 6-4
- response
 - definition 3-7
- response inquiry
 - built-in functions 6-7
- response statement 3-9
- response to
 - char 6-2, 6-8
 - interval 6-3
 - line 6-2, 6-8
 - low memory 6-3, 6-17, 6-19
 - object 6-2
 - object class 6-35
 - overview 6-2
 - start 6-2, 6-19
 - stimulus 6-2
 - termination 6-3, 6-17, 6-19
 - timeout 6-3
- response to char 14-13
- response to char statement 14-2, 14-7, 14-9, 15-9
- response to interval statement 11-29
- response to item
 - model 10-27

- response to item statement 6-6
- response to line 14-8, 14-13
- response to line statement 14-2, 14-7, 15-9
- response to low memory statement 11-34, 11-38
- response to object statement 6-4, 8-26
- response to start 11-11
- response to start statement 6-3
- response to stimulus
 - definition 14-29
- response to termination statement 6-16
- response to timeout statement 11-30
- responses 3-8
- resumable block
 - definition 6-11
 - description 6-17
- right of function 4-43, 7-55, 8-41
- rotating
 - pattern 7-37
 - specifications 7-37
 - text 7-39

S

- sample keyword 7-67
- sample programs
 - audio support 18-31
 - automatic interface 19-93
 - controlled interface 19-95
- save as program statement 11-2
- save program as statement 4-18, 11-10, 11-12
 - to save program in progress 11-15
 - with global and local values 11-15
- scalable font 5-50
- scaling
 - in image regions 7-67
 - pattern 7-36
- ScanScreen 20-21

- scatter graph 16-44
- screen
 - ancestry of 5-21
 - coordinate system 5-16
 - origin 5-17
 - resolution 5-17
- Screen Size 3-16, 7-56
- screen size declaration 3-6
- screen size Environment Declaration 3-16
- screen size statement 5-16, 5-21
- scroll attribute 10-6
- scroll bar attribute 3-16
- segment
 - copying contents of 8-43
 - in color header 8-60
 - in colored textual regions 8-55
 - text 8-17
 - textual region 8-10
 - with textual function 8-44
- selectability of function 4-43, 6-28, 7-62, 8-42
- selected line from function 4-43
- selected line function 4-45
- send statement 4-45, 14-2, 14-7, 24-4
 - errorlevel function 4-45
- sending/receiving non-string data in APPC 19-15
- sending/receiving string data in APPC 19-14
- Sense regions 7-72
- senseport 5-5
- separator definition
 - model of 10-13
- separators
 - defining 10-13
- serial number 1-3
- set command (OS/2) 3-14
- set low memory threshold
 - statement 11-34, 11-36
- set pointer to statement 15-2
- SetEmulatorWindow 20-50
- shape
 - border of 7-29
 - color of 7-29
- shape (*continued*)
 - solid or hollow 7-28
- shape statement 7-9, 7-15, 7-19
- shapes
 - graphics cursor 7-23
- shift keyword 10-15
- single-line entry fields 9-8
- size
 - considerations 3-21
 - inquiry 7-54
 - viewports 8-4
 - windows 8-4
- size attribute 7-4
- SKIP_CLAUSE 4-23
- sound 15-14
- spacepar 14-21
- SQL Support
 - access plans 17-2
 - case of characters 17-32
 - command line options 17-30
 - command types 17-2
 - configuration commands 17-7
 - direct commands 17-3
 - dynamic commands 17-26
 - dynamic commands
 - requirement 17-32
 - error handling 17-27
 - error message 17-32
 - graphic data types 17-33
 - increasing memory 17-32
 - input 17-2
 - input command 17-31
 - inquiry commands 17-21
 - isolation level 17-32
 - LONG VARCHAR 17-31
 - notes 17-31
 - output 17-2
 - output, using 17-31
 - records, number of 17-32
 - requirements 17-2
 - standard error 17-32
 - statement size 17-32
 - strings, customizing 17-31
 - white space 17-32
- SQL Support commands
 - bindddb 17-6

SQL Support commands (*continued*)

- bindfile 17-12
- display 17-22
- error 17-9
- exit 17-20
- fs 17-14
- info 17-11
- maxwidth 17-17
- maxwidth command 17-17
- names 17-23
- null 17-10
- output 17-13
- prompt 17-8
- rs 17-15
- rs command 17-15
- set 17-16
- set command 17-16
- startdb 17-4
- startq 17-18
- stopdb 17-5
- types 17-24
- verbose 17-19
- widths 17-25
- squeeze memory statement 11-35, 11-37
- stacked bar graph 16-46
- stacked line graph 16-42
- standalone program 25-2
- standard bar graph 16-44
- standard line graph 16-41
- start local 14-4
- start remote 14-5
- start statement 14-4
 - errorlevel function 4-45
- start statement start action 14-2
- startdb command 17-4
- starting
 - 3270 20-14
 - 5250 21-14, 21-17
- startq command 17-18
- statement 8-25
- statement(s)
 - variable 3-7
- static text 9-21
- stepped line graph 16-14, 16-42

stimulus

- declaration, audio 18-4
- declaration, in APPC 19-5
- events in APPC, responding to 19-4, 19-8
- events, audio, responding to 18-4
- stimulus declaration 14-29
- Stimulus libraries 14-29
- stimulus response 14-29
- stop all statement 14-6
- stop statement 14-2, 14-6
- stopbits 14-22
- stopdb command 17-5
- stopping
 - 3270 20-14
 - 5250 21-14, 21-17
- Stop3270 20-18
- string
 - finding 8-34
- string data, sending/receiving in APPC 19-14
- strings 25-5
- subroutine declaration 11-2
- subroutine is definition 11-2
- subroutine is statement 11-6
- subroutines 11-4
 - open and close 13-31
 - output 13-41
 - writing 25-3
- supported 7-63
 - bitmap file formats 7-63
- surface line graph 16-43
- suspend processing 11-31
- synchronization
 - 3270 20-14
 - 5250 21-15
- syntax
 - frame oriented 3-16
- system menu accelerators
 - use in fullscreen applications 3-21
- system menu attribute 3-16, 10-5
- system menu keyword 3-17
- system monospaced font 5-47

system proportional font 5-47

T

tab character 8-16, 8-17, 8-34

TAKE_CLAUSE 4-23

-terp 23-10, 24-3, 24-5

text

- aligning 7-45

- centering 7-45

- changing disposition of 14-16

- color of 7-46

- coloring 8-55, 8-58

- converting to uppercase 5-46

- deemphasizing 8-30

- displaying in graphical
objects 7-44

- displaying outline in a graphical
region 7-45

- emphasizing 8-29, 8-30, 8-55

- font of 7-46

- overwriting 8-20

- removing 8-28

- rotating 7-39

text align statement 7-45

text center statement 7-45

text cursor 5-35

- in textual region 5-37

- visibility of 8-30

text of function 4-43, 8-42

text of item function 4-44

text statement 7-9, 7-44

- font statement 7-46

textual built-in function 8-42

textual function 4-43, 8-43

- column keyword 8-43

- from keyword 8-43

- line keyword 8-43

- segment keyword 8-43

- thru keyword 8-43

- with extract from

- statement 8-46

- with insert statement 8-44, 8-45

textual line from function 4-43

textual region 8-5

- action statements 8-27

textual region (*continued*)

- add to statement 8-28

- addressing a character in 8-8

- ancestry 8-2, 8-38

- appending to a file 8-31

- attributes 8-4

- colored 8-5

- columns and lines in 8-46

- coordinate system 5-20, 8-8

- coordinate system of 8-2

- defining 8-1, 8-2

- definition 8-4

- drawing statements 8-2, 8-7

- finding text in 8-34

- fonts 8-3, 8-6, 8-33

- foreground color 5-11

- move statement 8-9

- overwriting 8-23

- overwriting a file with 8-32

- program, sample 8-49

- reading a file, colored 8-33

- reading to 8-32

- response definitions 8-26

- segment, colored 8-10

- senseport 8-26

- size 8-4

- text cursor 5-37, 8-9

- text cursor in 8-2

- viewport 8-4, 8-36

- window 8-4, 8-36

- window of 8-2

- window size 8-4

- window, changing size 8-38

- writing to a color-attributed
file 8-32

- writing to a file 8-30

textual region definition

- color keyword 8-54

- default keyword 8-54

- foreground keyword 8-54

- position keyword 8-54

textual regions

- Block (Text), colored 8-12

- built-in Functions 8-41

textual region, changing position

- viewport 8-37

- textual region, changing position
(*continued*)
 - window 8-39
- textual region, changing size
 - viewport 8-37
- textual region, reducing size
 - viewport 8-37
- three dimension b 16-10
- 3-D stacked bar graph
- three dimensional bar graph 16-10
- three-dimension 16-45
- 3-D Bar Graph
- three-dimension o 16-11
- 3-D overlapping bar graph 16-46
- time function 4-45
- timing
 - Program 11-29
- title bar attribute 3-16, 10-4
- title bar keyword 3-17
- tms rmn font 5-48
- to keyword 5-35
- top of function 4-43, 7-55, 8-41
- touchscreen 15-2
- touchscreen keyword 15-7
- tracing, enabling APPC 19-101
- turn trace action statement 24-3
- turn trace off statement 24-3, 24-5
- turn trace on statement 24-3, 24-5
- turn trace statement 24-5
- type conversions 4-53
- types command 17-24
- TypeString 20-48

U

- uncheck keyword 10-22
- update screen
 - wait 0 statement 11-32
- uppercase
 - conversion to 5-46
 - font 5-46
- utilities
 - printing 22-5
- utility subroutines, APPC 19-22
- U.S. code page 4-4

V

- value
 - boolean 11-22
 - compile-time 4-11
 - constants 4-16
 - literals 4-12
 - string, copying to a 8-43
 - type conversion 4-53
 - types 4-10
 - variables 4-17
- value in match clause 14-10
- variable
 - arrays 11-14
 - assigning value(s) to 4-17, 4-20
 - global 11-14
 - global and local 4-18
 - initializing 4-17
 - string, finding length of 4-19
- variable statement 4-17
- verbose command 17-19
- vertical scroll bar keyword 3-17
- VGA
 - fullscreen colors 5-9
- viewport 5-23
 - ancestry 5-30
 - change size statement 7-48
 - changing size of 7-48
 - graphical region 7-48
 - in graphical region 7-47
 - region 5-28
- viewport, changing position of
 - graphical region 7-50
- visibility
 - of text cursor 8-30
- visibility attribute 7-73
- visibility of function 4-43, 6-28, 7-62, 8-42
- visible keyword 5-8

W

- wait for statement 11-31
- wait statement 5-52, 11-31
- Watch 20-41

- WatchAndWait 20-43
- WatchForChar 20-39
- WatchForCursorAt 20-36
- WatchForCursorNotAt 20-37
- WatchForNotChar 20-40
- WatchForNoX 20-38
- wedge
 - color of 7-19
 - colored 7-19
 - degrees 7-18
 - limitations 7-19
 - solid 7-19
 - solid or hollow 7-19
 - statement 7-18
- wedge statement 7-9, 7-18
 - definition 7-18
- welcome.eal. 3-11
- while clause
 - in loop statement 11-21
- widths command 17-25
- window 5-22
 - ancestry 5-30
 - changing position of 7-52
 - changing size of 7-51
 - graphical region 7-51, 7-52
 - in graphical region 7-47
 - region 5-24
- window xposition of function 4-43, 7-58, 8-43
- window xsize of function 4-43, 7-57, 8-42
- window yposition of function 4-43, 7-58, 8-43
- window ysize of function 4-43, 7-58, 8-43
- windowed application
 - scaling 3-20
- windowed applications
 - architecture 3-21
- windowed vs fullscreen
 - applications 3-19
- with textual function
 - from keyword 9-14
 - thru keyword 9-14
- word 14-11

- word in match clause 14-9
- word wrap
 - in multiline entry field
 - definition 9-10
- work area cache 7-66
- write statement 7-70
 - append keyword 8-30, 8-31
 - colored file keywords 8-30
 - destructive 8-32
 - destructive keyword 8-30
 - file keyword 8-30
 - ioerror function 4-44
 - ioerror function with 8-31
- WriteField 20-56
- writing
 - DLLs 25-3
 - functions 25-3
 - guidelines 25-2
 - keyboard 25-2
 - libraries 25-1, 25-3
 - local applications 25-1
 - screen I/O 25-2
 - standalone program 25-2
 - subroutines 25-3
 - well-behave applications 25-2

X

- xcoord function 4-42, 6-7, 6-9, 8-26, 8-41
- xcursor of function 4-43, 7-56, 8-35, 8-41
- xmiddle of function 4-43, 7-55, 8-41
- XNumTicks 16-23
- xposition of function 4-43, 7-57, 8-41
- xsize of function 4-43, 7-57, 8-41
- XTicks 16-23

Y

- ycoord function 4-42, 6-7, 6-9, 8-26, 8-41
- ycursor of function 4-43, 7-56, 8-35, 8-41

- ymiddle of function 4-43, 7-56, 8-41
- YNumTicks 16-23
- yposition 7-57
- yposition of function 4-43, 8-41
- ysize of function 4-43, 7-57, 8-42
- YTicks 16-23

Z

- zooming
 - in 5-25
 - out 5-25

Numerics

- 3270 action routines
 - Connect 20-19
 - DefineWatch 20-35
 - Disconnect 20-20
 - EmulateAndWatch 20-53
 - Emulate3270 20-52
 - EnterString 20-49
 - FindLastNonBlankLine 20-29
 - GetCursorPosition 20-25
 - GetFieldAttributes 20-24
 - GetFieldLength 20-23
 - GetFieldNumber 20-26
 - GetFieldPosition 20-27
 - GetSessionInfo 20-62
 - GetSessionStatus 20-60
 - GetXstatus 20-57
 - Init3270 20-17
 - PressENTER 20-46
 - PressKey 20-45
 - PressPFKey 20-47
 - Print3270Screen 20-59
 - ReadColoredScreen 20-31
 - ReadField 20-22
 - ReadLine 20-28
 - ReadScreen 20-30
 - ScanScreen 20-21
 - SetEmulatorWindow 20-50
 - Stop3270 20-18
 - TypeString 20-48
 - Watch 20-41
 - WatchAndWait 20-43

- 3270 action routines (*continued*)
 - WatchForChar 20-39
 - WatchForCursorAt 20-36
 - WatchForCursorNotAt 20-37
 - WatchForNotChar 20-40
 - WatchForNoX 20-38
 - WriteField 20-56
- 3270 Support 20-19
 - A3270.EXE 20-2
 - buffer, examining 20-21
 - communicating with host 20-45
 - components 20-2
 - Connecting 20-19
 - controlling emulator
 - window 20-50
 - Disconnecting 20-19
 - error handling 20-64
 - error messages 20-66
 - error status variables 20-64
 - fields utility 20-2, 20-8
 - miscellaneous action
 - routines 20-56
 - module statement 20-3
 - M3270 action routines 20-13
 - M3270 module 20-2
 - program sample 20-4
 - program sample, screens 20-75
 - referencing M3270 20-3
 - required procedures 20-3
 - reserved constants 20-74
 - reserved variables 20-71
 - screen size 20-6
 - screen, field-orientated 20-6
 - screen, line-orientated 20-6
 - sessions 20-5
 - starting 20-17
 - stopping 20-17
 - synchronizing 20-33
- 5250 action routines
 - M52_CheckIndicators 21-56
 - M52_Connect 21-19
 - M52_DefineWatch 21-39
 - M52_Disconnect 21-20
 - M52_EnterString 21-54
 - M52_FindLastNonBlankLine 21-29
 - M52_GetCursorPosition 21-25

5250 action routines (*continued*)

- M52_GetFieldAttributes 21-24
- M52_GetFieldLength 21-23
- M52_GetFieldNumber 21-26
- M52_GetFieldPosition 21-27
- M52_GetSessionInfo 21-57
- M52_GiveUpTime 21-45
- M52_InformWhenIloff 21-34
- M52_Init 21-17
- M52_PressCMDkey 21-52
- M52_PressENTER 21-51
- M52_PressKey 21-49
- M52_PrintScreen 21-59
- M52_ReadColoredScreen 21-31
- M52_ReadField 21-22
- M52_ReadLine 21-28
- M52_ReadScreen 21-30
- M52_ScanScreen 21-21
- M52_Stop 21-18
- M52_TypeString 21-53
- M52_WaitForIloff 21-36
- M52_Watch 21-45
- M52_WatchAndWait 21-47
- M52_WatchCommandString 21-41
- M52_WatchForChar 21-43
- M52_WatchForCursorAt 21-40
- M52_WatchForCursorNotAt 21-41
- M52_WatchForIIOff 21-42
- M52_WatchForNotChar 21-44
- M52_WriteField 21-55

5250 Support

- action routines 21-13
- action routines, summary 21-14
- A5250.EXE 21-2
- communicating with host 21-49
- components 21-2
- connecting 21-19
- disconnecting 21-19
- error handling 21-60
- error messages 21-62
- examining the buffer 21-21
- field-oriented 21-6
- fields utility 21-3, 21-9
- Input Inhibited indicator 21-33
- line-oriented screens 21-6
- module statement 21-3

5250 Support (*continued*)

- M5250 module 21-2
- program sample 21-4
- referencing the module 21-3
- required procedures 21-3
- reserved constants 21-70
- reserved variables 21-67
- retrieving information 21-9
- session names 21-5
- session states 21-5
- starting 21-17
- stopping 21-17
- synchronizing via 3270 device
 - emulation 21-37
- synchronizing with
 - AS/400 21-34
- synchronizing with host 21-33
- using 5250 screens 21-71
- watch facility 21-37

Special Characters

- .CMD file 24-5
- .EAL file 3-12
- .EBI file 24-2

Continued from inside front cover

SUCH WARRANTIES ARE IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow the exclusion of implied warranties, so the above exclusion may not apply to you.

LIMITATION OF REMEDIES

IBM's entire liability and your exclusive remedy shall be as follows:

1. IBM will provide the warranty described in IBM's Statement of Limited Warranty. If IBM does not replace defective media or, if applicable, make the Program operate as warranted or replace the Program with a functionally equivalent Program, all as warranted you may terminate your license and your money will be refunded upon the return of all of your copies of the Program.
2. For any claim arising out of IBM's limited warranty, or for any other claim whatsoever related to the subject matter of this Agreement, IBM's liability for actual damages, regardless of the form of action, shall be limited to the greater of \$5,000 or the money paid to IBM, its Authorized Dealer or its approved supplier for the license for the Program that caused the damages or that is the subject matter of, or is directly related to, the cause of action. This limitation will not apply to claims for personal injury or damages to real or tangible personal property caused by IBM's negligence.
3. In no event will IBM be liable for any lost profits, lost savings, or any incidental damages or other conse-

quential damages, even if IBM, its Authorized Dealer or its approved supplier has been advised of the possibility of such damages, or for any claim by you based on a third party claim.

Some states do not allow the limitation or exclusion of incidental or consequential damages so the above limitation or exclusion may not apply to you.

GENERAL

You may terminate your license at any time by destroying all your copies of the Program or as otherwise described in this Agreement.

IBM may terminate your license if you fail to comply with the terms and conditions of this Agreement. Upon such termination you agree to destroy all your copies of the Program.

Any attempt to sublicense, rent, lease or assign, or, except as expressly provided herein, to transfer any copy of the Program is void.

You agree that you are responsible for payment of any taxes, including personal property taxes, resulting from this Agreement.

No action, regardless of form, arising out of this Agreement may be brought by either party more than two years after the cause of action has arisen except for breach of the provisions in the Section entitled "License" in which event four years shall apply.

This Agreement will be construed under the Uniform Commercial Code of the State of New York.

Z125-3301-02 4/87
Printed in U.S.A.



SC38-7034-01



Printed in U.S.A.